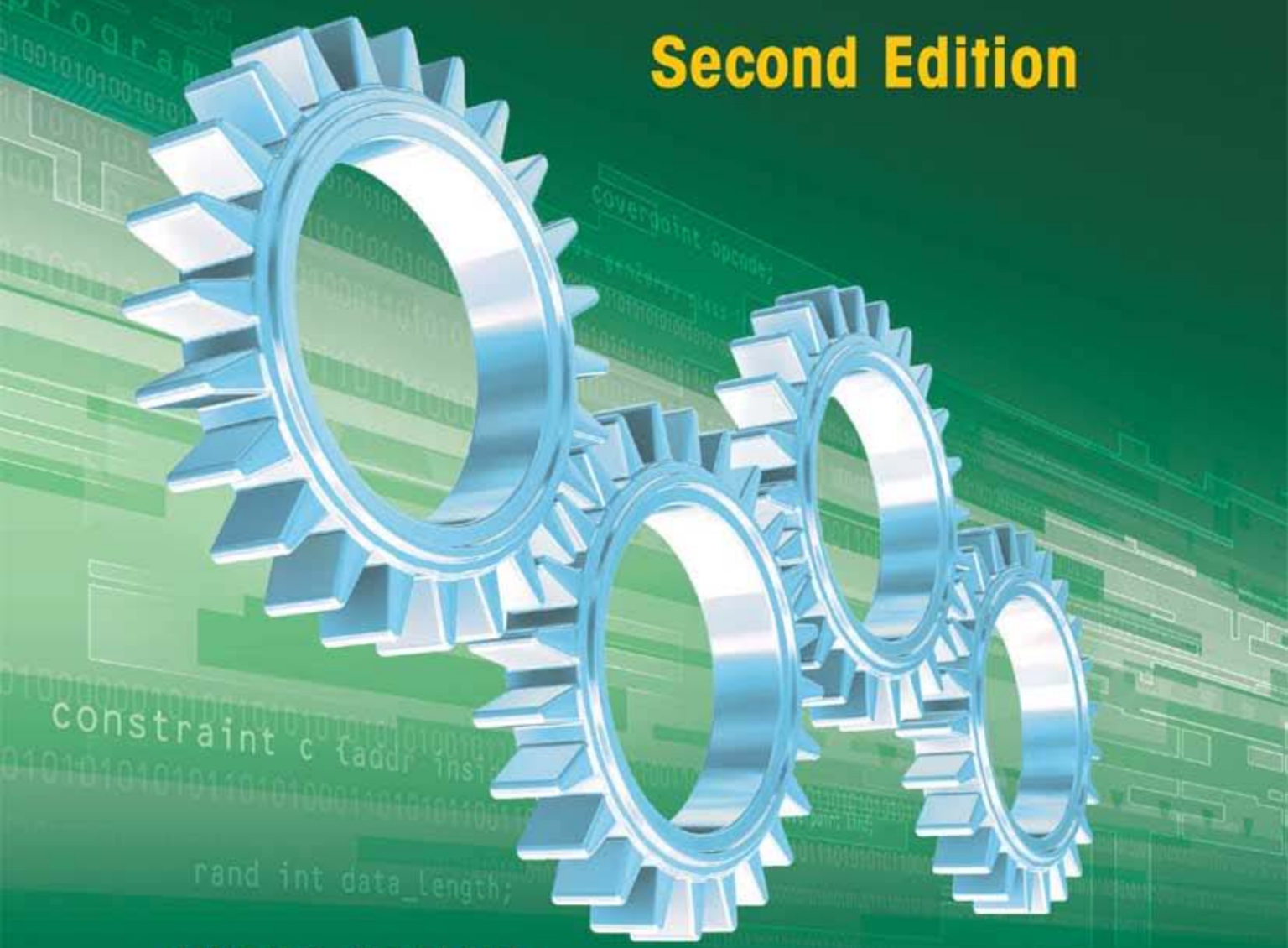
SystemVerilog for Verification

A Guide to Learning the Testbench Language Features

Second Edition



CHRIS SPEAR

Springer

SystemVerilog for Verification

A Guide to Learning the Testbench Language Features

Second Edition

Chris Spear

SystemVerilog for Verification

A Guide to Learning the Testbench Language Features

Second Edition

Springer

Chris Spear
Synopsys, Inc.
Marlboro, MA
USA

Library of Congress Control Number: 2008920031

ISBN 978-0-387-76529-7 e-ISBN 978-0-387-76530-3

Printed on acid-free paper.

©2008 Springer Science+Business Media, LLC
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or heareafter developed is forbidden. The use in this publication of trade names, trademarks, service marks and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.
While the advice and information in this book are believed to be true and accurate at the date of going to press, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

9 8 7 6 5 4 3 2 1

springer.com

This book is dedicated to my wonderful wife Laura,
whose patience during this project was invaluable,
and my children, Allie and Tyler, who kept me laughing.

Contents

List of Examples xiii
List of Figures xxv
List of Tables xxvii
Preface xxix
Acknowledgments xxxv

1. VERIFICATION GUIDELINES 1
- 1.1 The Verification Process 2
- 1.2 The Verification Methodology Manual 4
- 1.3 Basic Testbench Functionality 5
- 1.4 Directed Testing 5
- 1.5 Methodology Basics 7
- 1.6 Constrained-Random Stimulus 8
- 1.7 What Should You Randomize? 10
- 1.8 Functional Coverage 13
- 1.9 Testbench Components 14
- 1.10 Layered Testbench 15
- 1.11 Building a Layered Testbench 21
- 1.12 Simulation Environment Phases 22
- 1.13 Maximum Code Reuse 23
- 1.14 Testbench Performance 23
- 1.15 Conclusion 24

2. DATA TYPES 25
- 2.1 Built-In Data Types 25
- 2.2 Fixed-Size Arrays 28
- 2.3 Dynamic Arrays 34
- 2.4 Queues 36
- 2.5 Associative Arrays 38

2.6 Linked Lists 40
2.7 Array Methods 41
2.8 Choosing a Storage Type 46
2.9 Creating New Types with typedef 48
2.10 Creating User-Defined Structures 50
2.11 Type conversion 52
2.12 Enumerated Types 55
2.13 Constants 59
2.14 Strings 59
2.15 Expression Width 60
2.16 Conclusion 61

3. PROCEDURAL STATEMENTS AND ROUTINES 63
3.1 Procedural Statements 63
3.2 Tasks, Functions, and Void Functions 65
3.3 Task and Function Overview 65
3.4 Routine Arguments 66
3.5 Returning from a Routine 72
3.6 Local Data Storage 73
3.7 Time Values 75
3.8 Conclusion 77

4. CONNECTING THE TESTBENCH AND DESIGN 79
4.1 Separating the Testbench and Design 80
4.2 The Interface Construct 82
4.3 Stimulus Timing 88
4.4 Interface Driving and Sampling 96
4.5 Connecting It All Together 103
4.6 Top-Level Scope 104
4.7 Program – Module Interactions 106
4.8 SystemVerilog Assertions 107
4.9 The Four-Port ATM Router 109
4.10 The ref Port Direction 117
4.11 The End of Simulation 118
4.12 Directed Test for the LC3 Fetch Block 118
4.13 Conclusion 124

5. BASIC OOP 125
5.1 Introduction 125
5.2 Think of Nouns, not Verbs 126
5.3 Your First Class 126

5.4 Where to Define a Class 127
5.5 OOP Terminology 128
5.6 Creating New Objects 129
5.7 Object Deallocation 132
5.8 Using Objects 134
5.9 Static Variables vs. Global Variables 134
5.10 Class Methods 138
5.11 Defining Methods Outside of the Class 139
5.12 Scoping Rules 141
5.13 Using One Class Inside Another 144
5.14 Understanding Dynamic Objects 147
5.15 Copying Objects 151
5.16 Public vs. Local 157
5.17 Straying Off Course 157
5.18 Building a Testbench 158
5.19 Conclusion 159

6. RANDOMIZATION 161
6.1 Introduction 161
6.2 What to Randomize 162
6.3 Randomization in SystemVerilog 165
6.4 Constraint Details 167
6.5 Solution Probabilities 178
6.6 Controlling Multiple Constraint Blocks 182
6.7 Valid Constraints 183
6.8 In-line Constraints 184
6.9 The pre_randomize and post_randomize Functions 185
6.10 Random Number Functions 187
6.11 Constraints Tips and Techniques 187
6.12 Common Randomization Problems 193
6.13 Iterative and Array Constraints 195
6.14 Atomic Stimulus Generation vs. Scenario Generation 204
6.15 Random Control 207
6.16 Random Number Generators 209
6.17 Random Device Configuration 213
6.18 Conclusion 216

7. THREADS AND INTERPROCESS COMMUNICATION 217
7.1 Working with Threads 218
7.2 Disabling Threads 228
7.3 Interprocess Communication 232
7.4 Events 233
7.5 Semaphores 238
7.6 Mailboxes 240

7.7 Building a Testbench with Threads and IPC 253
7.8 Conclusion 257

8. ADVANCED OOP AND TESTBENCH GUIDELINES 259
8.1 Introduction to Inheritance 260
8.2 Blueprint Pattern 265
8.3 Downcasting and Virtual Methods 270
8.4 Composition, Inheritance, and Alternatives 274
8.5 Copying an Object 279
8.6 Abstract Classes and Pure Virtual Methods 282
8.7 Callbacks 284
8.8 Parameterized Classes 290
8.9 Conclusion 293

9. FUNCTIONAL COVERAGE 295
9.1 Coverage Types 298
9.2 Functional Coverage Strategies 301
9.3 Simple Functional Coverage Example 303
9.4 Anatomy of a Cover Group 305
9.5 Triggering a Cover Group 307
9.6 Data Sampling 310
9.7 Cross Coverage 319
9.8 Generic Cover Groups 325
9.9 Coverage Options 327
9.10 Analyzing Coverage Data 329
9.11 Measuring Coverage Statistics During Simulation 331
9.12 Conclusion 332

10. ADVANCED INTERFACES 333
10.1 Virtual Interfaces with the ATM Router 334
10.2 Connecting to Multiple Design Configurations 342
10.3 Procedural Code in an Interface 347
10.4 Conclusion 350

11. A COMPLETE SYSTEMVERILOG TESTBENCH 351
11.1 Design Blocks 351
11.2 Testbench Blocks 356
11.3 Alternate Tests 377
11.4 Conclusion 379

12. INTERFACING WITH C 381
12.1 Passing Simple Values 382

12.2 Connecting to a Simple C Routine 385
12.3 Connecting to C++ 393
12.4 Simple Array Sharing 398
12.5 Open arrays 400
12.6 Sharing Composite Types 404
12.7 Pure and Context Imported Methods 407
12.8 Communicating from C to SystemVerilog 407
12.9 Connecting Other Languages 418
12.10 Conclusion 419

References 421
Index 423

List of Code Samples

Sample 1.1	Driving the APB pins	16
Sample 1.2	A task to drive the APB pins	17
Sample 1.3	Low-level Verilog test	17
Sample 1.4	Basic transactor code	21
Sample 2.1	Using the logic type	26
Sample 2.2	Signed data types	27
Sample 2.3	Checking for 4-state values	27
Sample 2.4	Declaring fixed-size arrays	28
Sample 2.5	Declaring and using multidimensional arrays	28
Sample 2.6	Unpacked array declarations	29
Sample 2.7	Initializing an array	29
Sample 2.8	Using arrays with `for-` and `foreach-`loops	30
Sample 2.9	Initialize and step through a multidimensional array	30
Sample 2.10	Output from printing multidimensional array values	30
Sample 2.11	Printing a multidimensional array	31
Sample 2.12	Output from printing multidimensional array values	31
Sample 2.13	Array copy and compare operations	32
Sample 2.14	Using word and bit subscripts together	32
Sample 2.15	Packed array declaration and usage	33
Sample 2.16	Declaration for a mixed packed/unpacked array	33
Sample 2.17	Using dynamic arrays	35
Sample 2.18	Using a dynamic array for an uncounted list	35
Sample 2.19	Queue operations	37
Sample 2.20	Queue operations	37
Sample 2.21	Declaring, initializing, and using associative arrays	39
Sample 2.22	Using an associative array with a string index	40
Sample 2.23	Creating the sum of an array	41
Sample 2.24	Picking a random element from an associative array	42
Sample 2.25	Array locator methods: min, max, unique	43
Sample 2.26	Array locator methods: find	43
Sample 2.27	Declaring the iterator argument	43

Sample 2.28	Array locator methods	44
Sample 2.29	Sorting an array	44
Sample 2.30	Sorting an array of structures	45
Sample 2.31	A scoreboard with array methods	45
Sample 2.32	User-defined type-macro in Verilog	49
Sample 2.33	User-defined type in SystemVerilog	49
Sample 2.34	Definition of uint	49
Sample 2.35	User-defined array type	50
Sample 2.36	Creating a single pixel type	50
Sample 2.37	The pixel struct	50
Sample 2.38	Initializing a struct	51
Sample 2.39	Using typedef to create a union	51
Sample 2.40	Packed structure	52
Sample 2.41	Converting between int and real with static cast	53
Sample 2.42	Basic streaming operator	53
Sample 2.43	Converting between queues with streaming operator	54
Sample 2.44	Converting between a structure and array with streaming operators	55
Sample 2.45	A simple enumerated type	55
Sample 2.46	Enumerated types	56
Sample 2.47	Specifying enumerated values	56
Sample 2.48	Incorrectly specifying enumerated values	57
Sample 2.49	Correctly specifying enumerated values	57
Sample 2.50	Stepping through all enumerated members	58
Sample 2.51	Assignments between integers and enumerated types	58
Sample 2.52	Declaring a const variable	59
Sample 2.53	String methods	60
Sample 2.54	Expression width depends on context	61
Sample 3.1	New procedural statements and operators	64
Sample 3.2	Using break and continue while reading a file	64
Sample 3.3	Void function for debug	65
Sample 3.4	Ignoring a function's return value	65
Sample 3.5	Simple task without begin...end	66
Sample 3.6	Verilog-1995 routine arguments	66
Sample 3.7	C-style routine arguments	66
Sample 3.8	Verbose Verilog-style routine arguments	67
Sample 3.9	Routine arguments with sticky types	67
Sample 3.10	Passing arrays using ref and const	68
Sample 3.11	Using ref across threads	69
Sample 3.12	Function with default argument values	70
Sample 3.13	Using default argument values	70
Sample 3.14	Binding arguments by name	71
Sample 3.15	Original task header	71
Sample 3.16	Task header with additional array argument	71
Sample 3.17	Task header with additional array argument	71
Sample 3.18	Return in a task	72

Sample 3.19	Return in a function	72
Sample 3.20	Returning an array from a function with a typedef	73
Sample 3.21	Passing an array to a function as a ref argument	73
Sample 3.22	Specifying automatic storage in program blocks	74
Sample 3.23	Static initialization bug	75
Sample 3.24	Static initialization fix: use automatic	75
Sample 3.25	Static initialization fix: break apart declaration and initialization	75
Sample 3.26	Time literals and $timeformat	76
Sample 3.27	Time variables and rounding	77
Sample 4.1	Arbiter model using ports	81
Sample 4.2	Testbench using ports	82
Sample 4.3	Top-level netlist without an interface	82
Sample 4.4	Simple interface for arbiter	83
Sample 4.5	Arbiter using a simple interface	83
Sample 4.6	Testbench using a simple arbiter interface	84
Sample 4.7	Top module using a simple arbiter interface	84
Sample 4.8	Bad test module includes interface	85
Sample 4.9	Connecting an interface to a module that uses ports	85
Sample 4.10	Interface with modports	86
Sample 4.11	Arbiter model with interface using modports	86
Sample 4.12	Testbench with interface using modports	86
Sample 4.13	Arbiter model with interface using modports	87
Sample 4.14	Interface with a clocking block	90
Sample 4.15	Interface with a clocking block	91
Sample 4.16	Race condition between testbench and design	93
Sample 4.17	Testbench using interface with clocking block	95
Sample 4.18	Signal synchronization	97
Sample 4.19	Synchronous interface sample and drive from module	97
Sample 4.20	Testbench using interface with clocking block	98
Sample 4.21	Interface signal drive	99
Sample 4.22	Driving a synchronous interface	99
Sample 4.23	Interface signal drive	100
Sample 4.24	Bidirectional signals in a program and interface	101
Sample 4.25	Bad clock generator in program block	102
Sample 4.26	Good clock generator in module	103
Sample 4.27	Top module using a simple arbiter interface	103
Sample 4.28	Module with just port connections	103
Sample 4.29	Module with an interface	104
Sample 4.30	Top module connecting DUT and interface	104
Sample 4.31	Top-level scope for arbiter design	105
Sample 4.32	Cross-module references with $root	106
Sample 4.33	Checking a signal with an if-statement	107
Sample 4.34	Simple immediate assertion	107
Sample 4.35	Error from failed immediate assertion	107
Sample 4.36	Creating a custom error message in an immediate assertion	108

Sample 4.37	Error from failed immediate assertion	108
Sample 4.38	Creating a custom error message	108
Sample 4.39	Concurrent assertion to check for X/Z	109
Sample 4.40	ATM router model header without an interface	111
Sample 4.41	Top-level netlist without an interface	112
Sample 4.42	Verilog-1995 testbench using ports	113
Sample 4.43	Rx interface	115
Sample 4.44	Tx interface	115
Sample 4.45	ATM router model with interface using modports	116
Sample 4.46	Top-level netlist with interface	116
Sample 4.47	Testbench using an interface with a clocking block	117
Sample 4.48	A `final` block	118
Sample 4.49	Fetch block Verilog code	120
Sample 4.50	Fetch block interface	121
Sample 4.51	Fetch block directed test	122
Sample 4.52	Top level block for fetch testbench	124
Sample 5.1	Simple transaction class	127
Sample 5.2	Declaring and using a handle	129
Sample 5.3	Simple user-defined new() function	130
Sample 5.4	A new() function with arguments	130
Sample 5.5	Calling the right new() function	131
Sample 5.6	Allocating multiple objects	132
Sample 5.7	Creating multiple objects	133
Sample 5.8	Using variables and routines in an object	134
Sample 5.9	Class with a static variable	135
Sample 5.10	The class scope resolution operator	136
Sample 5.11	Static storage for a handle	137
Sample 5.12	Static method displays static variable	138
Sample 5.13	Routines in the class	139
Sample 5.14	Out-of-block method declarations	140
Sample 5.15	Out-of-body task missing class name	141
Sample 5.16	Name scope	142
Sample 5.17	Class uses wrong variable	143
Sample 5.18	Move class into package to find bug	143
Sample 5.19	Using this to refer to class variable	144
Sample 5.20	Statistics class declaration	145
Sample 5.21	Encapsulating the Statistics class	145
Sample 5.22	Using a typedef class statement	146
Sample 5.23	Passing objects	148
Sample 5.24	Bad transaction creator task, missing ref on handle	149
Sample 5.25	Good transaction creator task with ref on handle	149
Sample 5.26	Bad generator creates only one object	149
Sample 5.27	Good generator creates many objects	150
Sample 5.28	Using an array of handles	150
Sample 5.29	Copying a simple class with new	151

Sample 5.30	Copying a complex class with new operator	152
Sample 5.31	Simple class with copy function	153
Sample 5.32	Using a copy function	153
Sample 5.33	Complex class with deep copy function	154
Sample 5.34	Statistics class declaration	154
Sample 5.35	Copying a complex class with new operator	155
Sample 5.36	Transaction class with pack and unpack functions	156
Sample 5.37	Using the pack and unpack functions	156
Sample 5.38	Basic Transactor	159
Sample 6.1	Simple random class	165
Sample 6.2	Constraint without random variables	167
Sample 6.3	Constrained-random class	168
Sample 6.4	Bad ordering constraint	168
Sample 6.5	Result from incorrect ordering constraint	169
Sample 6.6	Constrain variables to be in a fixed order	169
Sample 6.7	Weighted random distribution with dist	170
Sample 6.8	Dynamically changing distribution weights	170
Sample 6.9	Random sets of values	171
Sample 6.10	Specifying minimum and maximum range with $	171
Sample 6.11	Inverted random set constraint	171
Sample 6.12	Random set constraint for an array	172
Sample 6.13	Equivalent set of constraints	172
Sample 6.14	Repeated values in inside constraint	173
Sample 6.15	Output from inside constraint operator and weighted array	173
Sample 6.16	Class to choose from an array of possible values	174
Sample 6.17	Choosing from an array of values	174
Sample 6.18	Using randc to choose array values in random order	175
Sample 6.19	Constraint block with implication operator	176
Sample 6.20	Constraint block with if-else operator	176
Sample 6.21	Bidirectional constraint	176
Sample 6.22	Expensive constraint with mod and unsized variable	177
Sample 6.23	Efficient constraint with bit extract	178
Sample 6.24	Class Unconstrained	178
Sample 6.25	Class with implication	179
Sample 6.26	Class with implication and constraint	180
Sample 6.27	Class with implication and solve...before	181
Sample 6.28	Using constraint_mode	183
Sample 6.29	Checking write length with a valid constraint	183
Sample 6.30	The randomize() with statement	184
Sample 6.31	Building a bathtub distribution	186
Sample 6.32	$urandom_range usage	187
Sample 6.33	Constraint with a variable bound	188
Sample 6.34	dist constraint with variable weights	188
Sample 6.35	rand_mode disables randomization of variables	189
Sample 6.36	Randomizing a subset of variables in a class	190

Sample 6.37	Using the implication constraint as a case statement	191
Sample 6.38	Turning constraints on and off with constraint_mode	191
Sample 6.39	Class with an external constraint	192
Sample 6.40	Program defining an external constraint	193
Sample 6.41	Signed variables cause randomization problems	194
Sample 6.42	Randomizing unsigned 32-bit variables	194
Sample 6.43	Randomizing unsigned 8-bit variables	194
Sample 6.44	Constraining dynamic array size	195
Sample 6.45	Random strobe pattern class	197
Sample 6.46	First attempt at sum constraint: bad_sum1	198
Sample 6.47	Program to try constraint with array sum	198
Sample 6.48	Output from bad_sum1	198
Sample 6.49	Second attempt at sum constraint: bad_sum2	198
Sample 6.50	Output from bad_sum2	199
Sample 6.51	Third attempt at sum constraint: bad_sum3	199
Sample 6.52	Output from bad_sum3	199
Sample 6.53	Fourth attempt at sum_constraint: bad_sum4	199
Sample 6.54	Output from bad_sum4	200
Sample 6.55	Simple foreach constraint: good_sum5	200
Sample 6.56	Output from good_sum5	200
Sample 6.57	Creating ascending array values with foreach	201
Sample 6.58	Creating unique array values with foreach	201
Sample 6.59	Creating unique array values with a randc helper class	202
Sample 6.60	Unique value generator	202
Sample 6.61	Class to generate a random array of unique values	203
Sample 6.62	Using the UniqueArray class	203
Sample 6.63	Constructing elements in a random array	204
Sample 6.64	Command generator using randsequence	205
Sample 6.65	Random control with randcase and $urandom_range	207
Sample 6.66	Equivalent constrained class	208
Sample 6.67	Creating a decision tree with randcase	209
Sample 6.68	Simple pseudorandom number generator	210
Sample 6.69	Test code before modification	212
Sample 6.70	Test code after modification	212
Sample 6.71	Ethernet switch configuration class	213
Sample 6.72	Building environment with random configuration	214
Sample 6.73	Simple test using random configuration	215
Sample 6.74	Simple test that overrides random configuration	215
Sample 7.1	Interaction of begin...end and fork...join	219
Sample 7.2	Output from begin...end and fork...join	220
Sample 7.3	Fork...join_none code	221
Sample 7.4	Fork...join_none output	221
Sample 7.5	Fork...join_any code	222
Sample 7.6	Output from fork...join_any	222
Sample 7.7	Generator / Driver class with a run task	223

Sample 7.8	Dynamic thread creation	224
Sample 7.9	Bad fork...join_none inside a loop	225
Sample 7.10	Execution of bad fork...join_none inside a loop	225
Sample 7.11	Automatic variables in a fork...join_none	226
Sample 7.12	Steps in executing automatic variable code	226
Sample 7.13	Automatic variables in a fork...join_none	227
Sample 7.14	Using wait fork to wait for child threads	227
Sample 7.15	Bug using shared program variable	228
Sample 7.16	Disabling a thread	229
Sample 7.17	Limiting the scope of a disable fork	230
Sample 7.18	Using disable label to stop threads	231
Sample 7.19	Using disable label to stop a task	232
Sample 7.20	Blocking on an event in Verilog	233
Sample 7.21	Output from blocking on an event	234
Sample 7.22	Waiting for an event	234
Sample 7.23	Output from waiting for an event	234
Sample 7.24	Waiting on event causes a zero delay loop	235
Sample 7.25	Waiting for an edge on an event	235
Sample 7.26	Passing an event into a constructor	236
Sample 7.27	Waiting for multiple threads with wait fork	237
Sample 7.28	Waiting for multiple threads by counting triggers	237
Sample 7.29	Waiting for multiple threads using a thread count	238
Sample 7.30	Semaphores controlling access to hardware resource	239
Sample 7.31	Bad generator creates only one object	241
Sample 7.32	Good generator creates many objects	242
Sample 7.33	Good driver receives transactions from mailbox	243
Sample 7.34	Exchanging objects using a mailbox: the Generator class	243
Sample 7.35	Exchanging objects using a mailbox: the Driver class	244
Sample 7.36	Exchanging objects using a mailbox: the program block	244
Sample 7.37	Bounded mailbox	245
Sample 7.38	Output from bounded mailbox	246
Sample 7.39	Producer–consumer without synchronization	247
Sample 7.40	Producer–consumer without synchronization output	248
Sample 7.41	Producer–consumer synchronized with bounded mailbox	249
Sample 7.42	Output from producer–consumer with bounded mailbox	249
Sample 7.43	Producer–consumer synchronized with an event	250
Sample 7.44	Producer–consumer synchronized with an event, continued	251
Sample 7.45	Output from producer–consumer with event	251
Sample 7.46	Producer–consumer synchronized with a mailbox	252
Sample 7.47	Output from producer–consumer with mailbox	253
Sample 7.48	Basic Transactor	254
Sample 7.49	Configuration class	255
Sample 7.50	Environment class	255
Sample 7.51	Basic test program	257
Sample 8.1	Base Transaction class	261

Sample 8.2	Extended Transaction class	262
Sample 8.3	Constructor with argument in an extended class	263
Sample 8.4	Driver class	264
Sample 8.5	Generator class	265
Sample 8.6	Generator class using blueprint pattern	267
Sample 8.7	Environment class	268
Sample 8.8	Simple test program using environment defaults	268
Sample 8.9	Injecting an extended transaction into testbench	269
Sample 8.10	Using inheritance to add a constraint	270
Sample 8.11	Base and extended class	271
Sample 8.12	Copying extended handle to base handle	271
Sample 8.13	Copying a base handle to an extended handle	272
Sample 8.14	Using $cast to copy handles	272
Sample 8.15	Transaction and BadTr classes	273
Sample 8.16	Calling class methods	273
Sample 8.17	Building an Ethernet frame with composition	276
Sample 8.18	Building an Ethernet frame with inheritance	277
Sample 8.19	Building a flat Ethernet frame	278
Sample 8.20	Base transaction class with a virtual copy function	279
Sample 8.21	Extended transaction class with virtual copy method	279
Sample 8.22	Base transaction class with copy_data function	280
Sample 8.23	Extended transaction class with copy_data function	281
Sample 8.24	Base transaction class with copy function	281
Sample 8.25	Extended transaction class with new copy function	282
Sample 8.26	Abstract class with pure virtual methods	283
Sample 8.27	Transaction class extends abstract class	283
Sample 8.28	Bodies for Transaction methods	284
Sample 8.29	Base callback class	286
Sample 8.30	Driver class with callbacks	286
Sample 8.31	Test using a callback for error injection	287
Sample 8.32	Simple scoreboard for atomic transactions	288
Sample 8.33	Test using callback for scoreboard	289
Sample 8.34	Stack using the int type	290
Sample 8.35	Parameterized class for a stack	291
Sample 8.36	Using the parameterized stack class	291
Sample 8.37	Parameterized generator class using blueprint pattern	292
Sample 8.38	Simple testbench using parameterized generator class	292
Sample 9.1	Incomplete D-flip flop model missing a path	299
Sample 9.2	Functional coverage of a simple object	303
Sample 9.3	Coverage report for a simple object	304
Sample 9.4	Coverage report for a simple object, 100% coverage	305
Sample 9.5	Functional coverage inside a class	307
Sample 9.6	Test using functional coverage callback	308
Sample 9.7	Callback for functional coverage	309
Sample 9.8	Cover group with a trigger	309

Sample 9.9	Module with SystemVerilog Assertion	309
Sample 9.10	Triggering a cover group with an SVA	310
Sample 9.11	Using auto_bin_max set to 2	311
Sample 9.12	Report with auto_bin_max set to 2	311
Sample 9.13	Using auto_bin_max for all cover points	312
Sample 9.14	Using an expression in a cover point	312
Sample 9.15	Defining bins for transaction length	313
Sample 9.16	Coverage report for transaction length	313
Sample 9.17	Specifying bin names	314
Sample 9.18	Report showing bin names	314
Sample 9.19	Specifying ranges with $	315
Sample 9.20	Conditional coverage – disable during reset	315
Sample 9.21	Using stop and start functions	316
Sample 9.22	Functional coverage for an enumerated type	316
Sample 9.23	Coverage report with enumerated types	316
Sample 9.24	Specifying transitions for a cover point	317
Sample 9.25	Wildcard bins for a cover point	317
Sample 9.26	Cover point with ignore_bins	318
Sample 9.27	Cover point with auto_bin_max and ignore_bins	318
Sample 9.28	Cover point with illegal_bins	318
Sample 9.29	Basic cross coverage	320
Sample 9.30	Coverage summary report for basic cross coverage	320
Sample 9.31	Specifying cross coverage bin names	321
Sample 9.32	Cross coverage report with labeled bins	322
Sample 9.33	Excluding bins from cross coverage	322
Sample 9.34	Specifying cross coverage weight	323
Sample 9.35	Cross coverage with bin names	324
Sample 9.36	Cross coverage with binsof	325
Sample 9.37	Mimicking cross coverage with concatenation	325
Sample 9.38	Simple argument	326
Sample 9.39	Pass-by-reference	326
Sample 9.40	Specifying per-instance coverage	327
Sample 9.41	Specifying comments for a cover group	328
Sample 9.42	Specifying comments for a cover group instance	328
Sample 9.43	Report all bins including empty ones	329
Sample 9.44	Specifying the coverage goal	329
Sample 9.45	Original class for transaction length	330
Sample 9.46	solve...before constraint for transaction length	330
Sample 10.1	Rx interface with clocking block	334
Sample 10.2	Tx interface with clocking block	334
Sample 10.3	Testbench using physical interfaces	335
Sample 10.4	Top level module with array of interfaces	336
Sample 10.5	Testbench using virtual interfaces	337
Sample 10.6	Testbench using virtual interfaces	337
Sample 10.7	Driver class using virtual interfaces	338

Sample 10.8	Test harness using an interface in the port list	340
Sample 10.9	Test with an interface in the port list	340
Sample 10.10	Top module with a second interface in the test's port list	340
Sample 10.11	Test with two interfaces in the port list	340
Sample 10.12	Test with virtual interface and XMR	341
Sample 10.13	Test harness without interfaces in the port list	341
Sample 10.14	Test harness with a second interface	341
Sample 10.15	Test with two virtual interfaces and XMRs	341
Sample 10.16	Interface for 8-bit counter	342
Sample 10.17	Counter model using X_if interface	343
Sample 10.18	Testbench using an array of virtual interfaces	343
Sample 10.19	Counter testbench using virtual interfaces	344
Sample 10.20	Driver class using virtual interfaces	345
Sample 10.21	Testbench using a typedef for virtual interfaces	346
Sample 10.22	Driver using a typedef for virtual interfaces	346
Sample 10.23	Testbench using an array of virtual interfaces	346
Sample 10.24	Testbench passing virtual interfaces with a port	347
Sample 10.25	Interface with tasks for parallel protocol	348
Sample 10.26	Interface with tasks for serial protocol	349
Sample 11.1	Top level module	353
Sample 11.2	Testbench program	354
Sample 11.3	CPU Management Interface	354
Sample 11.4	Utopia interface	355
Sample 11.5	Environment class header	356
Sample 11.6	Environment class methods	357
Sample 11.7	Callback class connects driver and scoreboard	360
Sample 11.8	Callback class connects monitor and scoreboard	360
Sample 11.9	Callback class connects the monitor and coverage	361
Sample 11.10	Environment configuration class	362
Sample 11.11	Cell configuration type	362
Sample 11.12	Configuration class methods	363
Sample 11.13	UNI cell format	363
Sample 11.14	NNI cell format	363
Sample 11.15	ATMCellType	364
Sample 11.16	UNI_cell definition	364
Sample 11.17	UNI_cell methods	365
Sample 11.18	UNI_generator class	368
Sample 11.19	driver class	368
Sample 11.20	Driver callback class	371
Sample 11.21	Monitor callback class	371
Sample 11.22	The Monitor class	371
Sample 11.23	The Scoreboard class	373
Sample 11.24	Functional coverage class	375
Sample 11.25	The CPU_driver class	376
Sample 11.26	Test with one cell	378

Sample 11.27	Test that drops cells using driver callback	379
Sample 12.1	SystemVerilog code calling C factorial routine	382
Sample 12.2	C factorial function	382
Sample 12.3	Changing the name of an imported function	383
Sample 12.4	Argument directions	383
Sample 12.5	C factorial routine with const argument	384
Sample 12.6	Importing a C math function	385
Sample 12.7	Counter method using a static variable	386
Sample 12.8	Testbench for an 7-bit counter with static storage	387
Sample 12.9	Counter method using instance storage	388
Sample 12.10	Testbench for an 7-bit counter with per-instance storage	389
Sample 12.11	Testbench for counter that checks for Z or X values	391
Sample 12.12	Counter method that checks for Z and X values	392
Sample 12.13	Counter class	393
Sample 12.14	Static methods and linkage	394
Sample 12.15	C++ counter communicating with methods	395
Sample 12.16	Static wrapper for C++ transaction level counter	396
Sample 12.17	Testbench for C++ model using methods	397
Sample 12.18	Testbench for C++ model using methods	398
Sample 12.19	C routine to compute Fibonacci series	398
Sample 12.20	Testbench for Fibonacci routine	399
Sample 12.21	C routine to compute Fibonacci series with 4-state array	399
Sample 12.22	Testbench for Fibonacci routine with 4-state array	399
Sample 12.23	Testbench code calling a C routine with an open array	400
Sample 12.24	C code using a basic open array	401
Sample 12.25	Testbench calling C code with multidimensional open array	402
Sample 12.26	C code with multidimensional open array	403
Sample 12.27	Testbench for packed open arrays	403
Sample 12.28	C code using packed open arrays	404
Sample 12.29	C code to share a structure	404
Sample 12.30	Testbench for sharing structure	405
Sample 12.31	Returning a string from C	406
Sample 12.32	Returning a string from a heap in C	406
Sample 12.33	Importing a pure function	407
Sample 12.34	Imported context tasks	407
Sample 12.35	Exporting a SystemVerilog function	408
Sample 12.36	Calling an exported SystemVerilog function from C	408
Sample 12.37	Output from simple export	408
Sample 12.38	SystemVerilog module for simple memory model	409
Sample 12.39	C code to read simple command file and call exported function	410
Sample 12.40	Command file for simple memory model	410
Sample 12.41	SystemVerilog module for memory model with exported tasks	411
Sample 12.42	C code to read command file and call exported function	411
Sample 12.43	Command file for simple memory model	412
Sample 12.44	Command file for exported methods with OOP memories	413

Sample 12.45	SystemVerilog module with memory model class	413
Sample 12.46	C code to call exported tasks with OOP memory	414
Sample 12.47	Second module for simple export example	415
Sample 12.48	Output from simple example with two modules	416
Sample 12.49	C code getting and setting context	416
Sample 12.50	Modules calling methods that get and set context	417
Sample 12.51	Output from svSetScope code	418
Sample 12.52	SystemVerilog code calling C wrapper for Perl	418
Sample 12.53	C wrapper for Perl script	419
Sample 12.54	Perl script called from C and SystemVerilog	419

List of Figures

Figure 1.1	Directed test progress over time	6
Figure 1.2	Directed test coverage	6
Figure 1.3	Constrained-random test progress over time vs. directed testing	8
Figure 1.4	Constrained-random test coverage	9
Figure 1.5	Coverage convergence	9
Figure 1.6	Test progress with and without feedback	13
Figure 1.7	The testbench – design environment	15
Figure 1.8	Testbench components	15
Figure 1.9	Signal and command layers	18
Figure 1.10	Testbench with functional layer added	18
Figure 1.11	Testbench with scenario layer added	19
Figure 1.12	Full testbench with all layers	20
Figure 1.13	Connections for the driver	21
Figure 2.1	Unpacked array storage	29
Figure 2.2	Packed array layout	33
Figure 2.3	Packed array bit layout	34
Figure 2.4	Associative array	38
Figure 4.1	The testbench – design environment	79
Figure 4.2	Testbench – Arbiter without interfaces	81
Figure 4.3	An interface straddles two modules	83
Figure 4.4	Main regions inside a SystemVerilog time step	94
Figure 4.5	A clocking block synchronizes the DUT and testbench	96
Figure 4.6	Sampling a synchronous interface	98
Figure 4.7	Driving a synchronous interface	100
Figure 4.8	Testbench – ATM router diagram without interfaces	110
Figure 4.9	Testbench – router diagram with interfaces	114
Figure 4.10	LD3 Microcontroller fetch block	119
Figure 5.1	Handles and objects after allocating multiple objects	132
Figure 5.2	Static variables in a class	135
Figure 5.3	Contained objects	144
Figure 5.4	Handles and objects across methods	147

Figure 5.5	Objects and handles before copy with the new operator	152
Figure 5.6	Objects and handles after copy with the new operator	152
Figure 5.7	Objects and handles after copy with the new operator	153
Figure 5.8	Objects and handles after deep copy	155
Figure 5.9	Layered testbench	158
Figure 6.1	Building a bathtub distribution	186
Figure 6.2	Random strobe waveforms	196
Figure 6.3	Sharing a single random generator	210
Figure 6.4	First generator uses additional values	211
Figure 6.5	Separate random generators per object	211
Figure 7.1	Testbench environment blocks	218
Figure 7.2	Fork...join blocks	219
Figure 7.3	Fork...join block	220
Figure 7.4	Fork...join block diagram	230
Figure 7.5	A mailbox connecting two transactors	241
Figure 7.6	A mailbox with multiple handles to one object	242
Figure 7.7	A mailbox with multiple handles to multiple objects	242
Figure 7.8	Layered testbench with environment	254
Figure 8.1	Simplified layered testbench	260
Figure 8.2	Base Transaction class diagram	261
Figure 8.3	Extended Transaction class diagram	262
Figure 8.4	Blueprint pattern generator	266
Figure 8.5	Blueprint generator with new pattern	266
Figure 8.6	Simplified extended transaction	271
Figure 8.7	Multiple inheritance problem	278
Figure 8.8	Callback flow	285
Figure 9.1	Coverage convergence	296
Figure 9.2	Coverage flow	297
Figure 9.3	Bug rate during a project	300
Figure 9.4	Coverage comparison	302
Figure 9.5	Uneven probability for transaction length	330
Figure 9.6	Even probability for transaction length with solve...before	330
Figure 10.1	Router and testbench with interfaces	336
Figure 11.1	The testbench – design environment	352
Figure 11.2	Block diagram for the squat design	352
Figure 12.1	Storage of a 40-bit 2-state variable	390
Figure 12.2	Storage of a 40-bit 4-state variable	391

List of Tables

Table 1.	Book icons	xxxiii
Table 4.1.	Primary SystemVerilog scheduling regions	94
Table 6.1.	Solutions for bidirectional constraint	177
Table 6.2.	Solutions for `Unconstrained` class	178
Table 6.3.	Solutions for `Imp1` class	179
Table 6.4.	Solutions for `Imp2` class	180
Table 6.5.	Solutions for `solve x before y` constraint	181
Table 6.6.	Solutions for `solve y before x` constraint	181
Table 8.1.	Comparing inheritance to composition	275
Table 12.1.	Data types mapping between SystemVerilog and C	384
Table 12.2.	4-state bit encoding	390
Table 12.3.	Open array query functions	401
Table 12.4.	Open array locator functions	402

Preface

What is this book about?

This book is the first one you should read to learn the SystemVerilog verification language constructs. It describes how the language works and includes many examples on how to build a basic coverage-driven, constrained-random layered testbench using Object-Oriented Programming (OOP). The book has many guidelines on building testbenches, which help show why you want to use classes, randomization, and functional coverage. Once you have learned the language, pick up some of the methodology books listed in the References section for more information on building a testbench.

Who should read this book?

If you create testbenches, you need this book. If you have only written tests using Verilog or VHDL and want to learn SystemVerilog, this book shows you how to move up to the new language features. Vera and Specman users can learn how one language can be used for both design and verification. You may have tried to read the SystemVerilog Language Reference Manual (LRM) but found it loaded with syntax but no guidelines on which construct to choose.

I wrote this book because, like many of my customers, I spent much of my career using procedural languages such as C and Verilog to write tests, and had to relearn everything when OOP verification languages came along. I made all the typical mistakes, and wrote this book so that you won't have to repeat them.

Before reading this book, you should be comfortable with Verilog-1995. Knowledge of Verilog-2001, SystemVerilog design constructs, or SystemVerilog Assertions is not required.

What is new in the second edition?

This new edition of SystemVerilog for Verification has many improvements over the first edition that was published in 2006.

- The anticipated 2008 version of the SystemVerilog Language Reference Manual (LRM) has many changes, both large and small. This book tries to include the latest relevant information.
- Many readers asked me for more details on SystemVerilog concepts. Almost all of these conversations have been incorporated into this book as expanded explanations and code samples. Starting with Chap. 2, nearly every paragraph and example has been rewritten, revised, or just tweaked. There are over 50 new pages in the original ten chapters, and over 70 new examples. In all, the new edition is almost 1/3 larger than the original.
- You asked for more examples, especially large ones. This edition has a directed testbench at the end of Chap. 4, and complete constrained random testbench in Chap. 11.
- Not all testbench code is written in SystemVerilog, and so I added Chap. 12 to show how to connect C and C++ code to SystemVerilog with the Direct Programming Interface.
- Most engineers read a book starting with the index, and so I doubled the number of entries. We also love cross references, and so I have added more so that you can read the book nonlinearly.
- Lastly, a big thanks to all the readers who spotted mistakes in the first edition, from poor grammar to code that was obviously written on the morning after a 18-hour flight from Asia to Boston. This edition has been checked and reviewed many times over, but once again, all mistakes are mine.

Why was SystemVerilog created?

In the late 1990s, the Verilog Hardware Description Language (HDL) became the most widely used language for describing hardware for simulation and synthesis. However, the first two versions standardized by the IEEE (1364-1995 and 1364-2001) had only simple constructs for creating tests. As design sizes outgrew the verification capabilities of the language, commercial Hardware Verification Languages (HVL) such as OpenVera and *e* were created. Companies that did not want to pay for these tools instead spent hundreds of man-years creating their own custom tools.

This productivity crisis (along with a similar one on the design side) led to the creation of Accellera, a consortium of EDA companies and users who wanted to create the next generation of Verilog. The donation of the OpenVera language formed the basis for the HVL features of SystemVerilog. Accellera's goal was met in November 2005 with the adoption of the IEEE standard P1800-2005 for SystemVerilog, IEEE (2005).

Importance of a unified language

Verification is generally viewed as a fundamentally different activity from design. This split has led to the development of narrowly focused language for verification and to the bifurcation of engineers into two largely independent disciplines. This specialization has created substantial bottlenecks in terms of communication between the two groups. SystemVerilog addresses this issue with its capabilities for both camps. Neither team has to give up any capabilities it needs to be successful, but the unification of both syntax and semantics of design and verification tools improves communication. For example, while a design engineer may not be able to write an object-oriented testbench environment, it is fairly straightforward to read such a test and understand what is happening, enabling both the design and verification engineers to work together to identify and fix problems. Likewise, a designer understands the inner workings of his or her block, and is the best person to write assertions about it, but a verification engineer may have a broader view needed to create assertions between blocks.

Another advantage of including the design, testbench, and assertion constructs in a single language is that the testbench has easy access to all parts of the environment without requiring specialized APIs. The value of an HVL is its ability to create high-level, flexible tests, not its loop constructs or declaration style. SystemVerilog is based on the Verilog constructs that engineers have used for decades.

Importance of methodology

There is a difference between learning the syntax of a language and learning how to use a tool. This book focuses on techniques for verification using constrained-random tests that use functional coverage to measure progress and direct the verification. As the chapters unfold, language and methodology features are shown side by side. For more on methodology, see Bergeron et al. (2006).

The most valuable benefit of SystemVerilog is that it allows the user to construct reliable, repeatable verification environments, in a consistent syntax, that can be used across multiple projects.

Comparing SystemVerilog and SystemC for high-level design

Now that SystemVerilog incorporates Object-Oriented Programming, dynamic threads, and interprocess communication, it can be used for system design. When talking about the applications for SystemVerilog, the IEEE standard mentions architectural modeling before design, assertions, and test. SystemC can also be used for architectural modeling.

There are several major differences between SystemC and SystemVerilog:

- SystemVerilog provides one modeling language. You do not have to learn C++ and the Standard Template Library to create your models.
- SystemVerilog simplifies top–down design. You can create your system models in SystemVerilog and then refine each block to the next lower level. The original system-level models can be reused as reference models.
- Software developers want a free or low-cost hardware simulator that is fast. You can create high-performance transaction-level models in both SystemC and SystemVerilog. SystemVerilog simulators require a license that a software developer may not want to pay for. SystemC can be free, but only if all your models are available in SystemC.

Overview of the book

The SystemVerilog language includes features for design, verification, assertions, and more. This book focuses on the constructs used to verify a design. There are many ways to solve a problem using SystemVerilog. This book explains the trade-offs between alternative solutions.

Chapter 1, *Verification Guidelines*, presents verification techniques to serve as a foundation for learning and using the SystemVerilog language. These guidelines emphasize coverage-driven random testing in a layered testbench environment.

Chapter 2, *Data Types*, covers the new SystemVerilog data types such as arrays, structures, enumerated types, and packed variables.

Chapter 3, *Procedural Statements and Routines*, shows the new procedural statements and improvements for tasks and functions.

Chapter 4, *Connecting the Testbench and Design*, shows the new SystemVerilog verification constructs, such as program blocks, interfaces, and clocking blocks, and how they are used to build your testbench and connect it to the design under test.

Chapter 5, *Basic OOP*, is an introduction to Object-Oriented Programming, explaining how to build classes, construct objects, and use handles.

Chapter 6, *Randomization*, shows you how to use SystemVerilog's constrained-random stimulus generation, including many techniques and examples.

Chapter 7, *Threads and Interprocess Communication*, shows how to create multiple threads in your testbench, use interprocess communication to exchange data between these threads and synchronize them.

Chapter 8, *Advanced OOP and Testbench Guidelines*, shows how to build a layered testbench with OOP so that the components can be shared by all tests.

Chapter 9, *Functional Coverage*, explains the different types of coverage and how you can use functional coverage to measure your progress as you follow a verification plan.

Chapter 10, *Advanced Interfaces*, shows how to use virtual interfaces to simplify your testbench code, connect to multiple design configurations, and create interfaces with procedural code so that your testbench and design can work at a higher level of abstraction.

Chapter 11, *A Complete SystemVerilog Testbench*, shows a constrained random testbench using the guidelines shown in Chap. 8. Several tests are shown to demonstrate how you can easily extend the behavior of a testbench without editing the original code, which always carries the risk of introducing new bugs.

Chapter 12, *Interfacing with C*, describes how to connect your C or C++ Code to SystemVerilog using the Direct Programming Interface.

Icons used in this book

Table 1. Book icons

	Shows verification methodology to guide your usage of SystemVerilog testbench features
	Shows common coding mistakes

Final comments

If you would like more information on SystemVerilog and Verification, you can find many resources at `http://chris.spear.net/systemverilog`

This site has the source code for many of the examples in this book. All of the examples have been verified with Synopsys' Chronologic VCS 2005.06, 2006.06, and 2008.03. The SystemVerilog Language Reference Manual covers hundreds of new features. I have concentrated on constructs useful for verification and implemented in VCS. It is better to have verified examples than to show all language features and thus risk having incorrect code. Speaking of mistakes, if you think you have found a mistake, please check my web site for the Errata page. If you are the first to find any mistake in a chapter, I will send you a free, autographed book.

CHRIS SPEAR
Synopsys, Inc.
chris@spear.net

Acknowledgments

Few books are the creation of a single person. I want to thank all the people who spent countless hours helping me learn SystemVerilog and reviewing the book that you now hold in your hand. I especially thank all the people at Synopsys for their help, including all my patient managers.

A big thanks to Shalom Bresticker, James Chang, David Lee, Ronald Mehler, Mike Mintz, Tim Pylant, Stuart Sutherland, and Tuan Tran, who reviewed some very rough drafts and inspired many improvements. However, the mistakes are all mine.

Janick Bergeron provided inspiration, innumerable verification techniques, and top-quality reviews. Without his guidance, this book would not exist.

Alex Potapov, Horia Toma, and the VCS R&D team always showed patience with my questions and provided valuable insight on SystemVerilog features.

Will Sherwood inspired me to become a verification engineer, and taught me new ways to break things.

The following people pointed out mistakes in the first edition, and made valuable suggestions on areas where the book could be improved: Dan Abate, Steve Barrett, Mike Blake, Shalom Bresticker, John Brooks, Heath Chambers, Keith Chan, Luke Chang, Haihui Chen, Gunther Clasen, Hashem Heidaragha, Stefan Kruepe, Jimnan Kuo, Jim Lewis, Daguang Liu, Victor Lopez, Michael Macheski, Chris Macionski, Mike Mintz, Thinh Ngo, John Nolan, Ben Rahardja, Afroza Rahman, Chandrasekar Rajanayagam, Jonathan Schmidt, Chandru Sippy, Dave Snogles, Tuan Tran, Robin van Malenhorst, Hugh Walsh, Larry Widigen, and Chunlin Zhang.

Jenny Bagdigian made sure I dotted my t's and crossed my i's. See you at Carnival!

United Airlines always had a quiet place to work and plenty of snacks. "Chicken or pasta?"

Lastly, a big thanks to Jay Mcinerney for his brash pronoun usage.

All trademarks and copyrights are the property of their respective owners.

Chapter 1

Verification Guidelines

"Some believed we lacked the programming language to describe your perfect world..."
(The Matrix, 1999)

Imagine that you are given the job of building a house for someone. Where should you begin? Do you start by choosing doors and windows, picking out paint and carpet colors, or selecting bathroom fixtures? Of course not! First you must consider how the owners will use the space, and their budget, so that you can decide what type of house to build. Questions you should consider are Do they enjoy cooking and want a high-end kitchen, or will they prefer watching movies in their home theater room and eating takeout pizza? Do they want a home office or an extra bedroom? Or does their budget limit them to a more modest house?

Before you start to learn details of the SystemVerilog language, you need to understand how you plan to verify your particular design and how this influences the testbench structure. Just as all houses have kitchens, bedrooms, and bathrooms, all testbenches share some common structure of stimulus generation and response checking. This chapter introduces a set of guidelines and coding styles for designing and constructing a testbench that meets your particular needs. These techniques use some of the same concepts that are shown in the *Verification Methodology Manual for SystemVerilog* (VMM), Bergeron et al. (2006), but without the base classes.

The most important principle you can learn as a verification engineer is "Bugs are good." Don't shy away from finding the next bug, do not hesitate to ring a bell each time you uncover one, and furthermore, always keep track of each bug found. The entire project team assumes there are bugs in the design, so that each bug found before tape-out is one fewer that ends up in the customer's hands. You need to be as

devious as possible, twisting and torturing the design to extract all possible bugs now, while they are still easy to fix. Don't let the designers steal all the glory – without your craft and cunning, the design might never work!

This book assumes you already know the Verilog language and want to learn the SystemVerilog Hardware Verification Language (HVL). Some of the typical features of an HVL that distinguish it from a Hardware Description Language such as Verilog or VHDL are

- Constrained-random stimulus generation
- Functional coverage
- Higher-level structures, especially object-oriented programming
- Multithreading and interprocess communication
- Support for HDL types such as Verilog's 4-state values
- Tight integration with event-simulator for control of the design

There are many other useful features, but these allow you to create testbenches at a higher level of abstraction than you are able to achieve with an HDL or a programming language such as C.

1.1 The Verification Process

What is the goal of verification? If you answered, "Finding bugs," you are only partly correct. The goal of hardware design is to create a device that performs a particular task, such as a DVD player, network router, or radar signal processor, based on a design specification. Your purpose as a verification engineer is to make sure the device can accomplish that task successfully – that is, the design is an accurate representation of the specification. Bugs are what you get when there is a discrepancy. The behavior of the device when used outside of its original purpose is not your responsibility, although you want to know where those boundaries lie.

The process of verification parallels the design creation process. A designer reads the hardware specification for a block, interprets the human language description, and creates the corresponding logic in a machine-readable form, usually RTL code. To do this, he or she needs to understand the input format, the transformation function, and the format of the output. There is always ambiguity in this interpretation, perhaps because of ambiguities in the original document, missing details, or conflicting descriptions. As a verification engineer, you must also read the hardware specification, create the verification plan, and then follow it to build tests showing the RTL code correctly implements the features.

By having more than one person perform the same interpretation, you have added redundancy to the design process. As the verification engineer, your job is to read the

same hardware specifications and make an independent assessment of what they mean. Your tests then exercise the RTL to show that it matches your interpretation.

1.1.1 Testing at Different Levels

What types of bugs are lurking in the design? The easiest ones to detect are at the block level, in modules created by a single person. Did the ALU correctly add two numbers? Did every bus transaction successfully complete? Did all the packets make it through a portion of a network switch? It is almost trivial to write directed tests to find these bugs, as they are contained entirely within one block of the design.

After the block level, the next place to look for discrepancies is at boundaries between blocks. Interesting problems arise when two or more designers read the same description yet have different interpretations. For a given protocol, what signals change and when? The first designer builds a bus driver with one view of the specification, while a second builds a receiver with a slightly different view. Your job is to find the disputed areas of logic and maybe even help reconcile these two different views.

To simulate a single design block, you need to create tests that generate stimuli from all the surrounding blocks – a difficult chore. The benefit is that these low-level simulations run very fast. However, you may find bugs in both the design and testbench, as the latter will have a great deal of code to provide stimuli from the missing blocks. As you start to integrate design blocks, they can stimulate each other, reducing your workload. These multiple block simulations may uncover more bugs, but they also run slower.

At the highest level of the DUT, the entire system is tested, but the simulation performance is greatly reduced. Your tests should strive to have all blocks performing interesting activities concurrently. All I/O ports are active, processors are crunching data, and caches are being refilled. With all this action, data alignment and timing bugs are sure to occur.

At this level you are able to run sophisticated tests that have the DUT executing multiple operations concurrently so that as many blocks as possible are active. What happens if an MP3 player is playing music and the user tries to download new music from the host computer? Then, during the download, the user presses several of the buttons on the player? You know that when the real device is being used, someone is going to do all this, and so why not try it out before it is built? This testing makes the difference between a product that is seen as easy to use and one that repeatedly locks up.

Once you have verified that the DUT performs its designated functions correctly, you need to see how it operates when there are errors. Can the design handle a partial transaction, or one with corrupted data or control fields? Just trying to enumerate all the possible problems is difficult, not to mention determining how the design should recover from them. Error injection and handling can be the most challenging part of verification.

As the design abstraction gets higher, so does the verification challenge. You can show that individual cells flow through the blocks of an ATM router correctly, but what if there are streams of different priority? Which cell should be chosen next is not always obvious at the highest level. You may have to analyze the statistics from thousands of cells to see if the aggregate behavior is correct.

One last point, you can never prove there are no bugs left, and so you need to constantly come up with new verification tactics.

1.1.2 The Verification Plan

The verification plan is closely tied to the hardware specification and contains a description of what features need to be exercised and the techniques to be used. These steps may include directed or random testing, assertions, HW/SW co-verification, emulation, formal proofs, and use of verification IP. For a more complete discussion on verification see Bergeron (2006).

1.2 The Verification Methodology Manual

This book in your hands draws heavily upon the VMM that has its roots in a methodology developed by Janick Bergeron and others at Qualis Design. They started with industry-standard practices and refined them based on their experience on many projects. VMM's techniques were originally developed for use with the OpenVera language and were extended in 2005 for SystemVerilog. VMM and its predecessor, the Reference Verification Methodology for Vera, have been used successfully to verify a wide range of hardware designs, from networking devices to processors. This book uses many of the same concepts.

This books serves as a user guide for the SystemVerilog language. It describes the languages many constructs and provides guidelines for choosing the ones best suited to your needs. If you are new to verification, have little experience with object-oriented programming, or are unfamiliar with constrained-random tests, this book can show you the right path to choose. Once you are familiar with them, you will find the VMM to be an easy step up.

So why doesn't this book teach you VMM? Like any advanced tool, VMM was designed for use by an experienced user, and excels on difficult problems. Are you in charge of verifying a 100 million-gate design with many communication protocols, complex error handling, and a library of IP? If so, VMM is the right tool for the job. However, if you are working on smaller modules, with a single protocol, you may not need such a robust methodology. Just remember that your block is part of a larger system; VMM-compliant code is reusable both during a project and on later designs. Remember that the cost of verification goes beyond your immediate project.

The VMM has a set of base classes for data and environment, utilities for managing log files and interprocess communication, and much more. This book is an introduction to SystemVerilog and shows the techniques and tricks that go into these classes and utilities, giving you insight into their construction.

1.3 Basic Testbench Functionality

The purpose of a testbench is to determine the correctness of the design under test (DUT). This is accomplished by the following steps.

- Generate stimulus
- Apply stimulus to the DUT
- Capture the response
- Check for correctness
- Measure progress against the overall verification goals

Some steps are accomplished automatically by the testbench, while others are manually determined by you. The methodology you choose determines how the preceding steps are carried out.

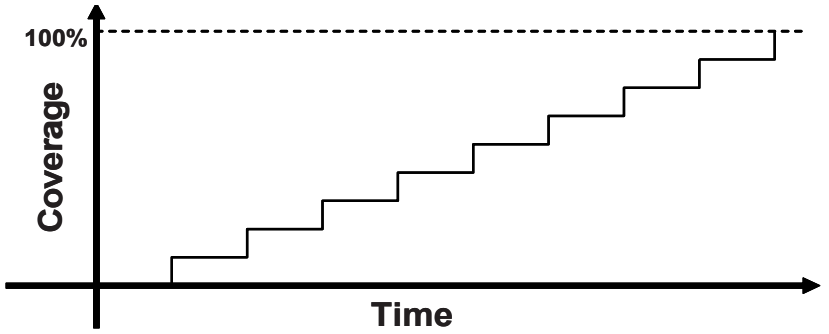
1.4 Directed Testing

Traditionally, when faced with the task of verifying the correctness of a design, you probably used directed tests. Using this approach, you look at the hardware specification and write a verification plan with a list of tests, each of which concentrated on a set of related features. Armed with this plan, you write stimulus vectors that exercise these features in the DUT. You then simulate the DUT with these vectors and manually review the resulting log files and waveforms to make sure the design does what you expect. Once the test works correctly, you check it off in the verification plan and move to the next one.

This incremental approach makes steady progress, which is always popular with managers who want to see a project making headway. It also produces almost immediate results, since little infrastructure is needed when you are guiding the creation of every stimulus vector. Given ample time and staffing, directed testing is sufficient to verify many designs.

Figure 1-1 shows how directed tests incrementally cover the features in the verification plan. Each test is targeted at a very specific set of design elements. If you had enough time, you could write all the tests needed for 100% coverage of the entire verification plan.

Figure 1-1 Directed test progress over time



What if you do not have the necessary time or resources to carry out the directed testing approach? As you can see, while you may always be making forward progress, the slope remains the same. When the design complexity doubles, it takes twice as long to complete or requires twice as many people to implement it. Neither of these situations is desirable. You need a methodology that finds bugs faster in order to reach the goal of 100% coverage.

Figure 1-2 Directed test coverage

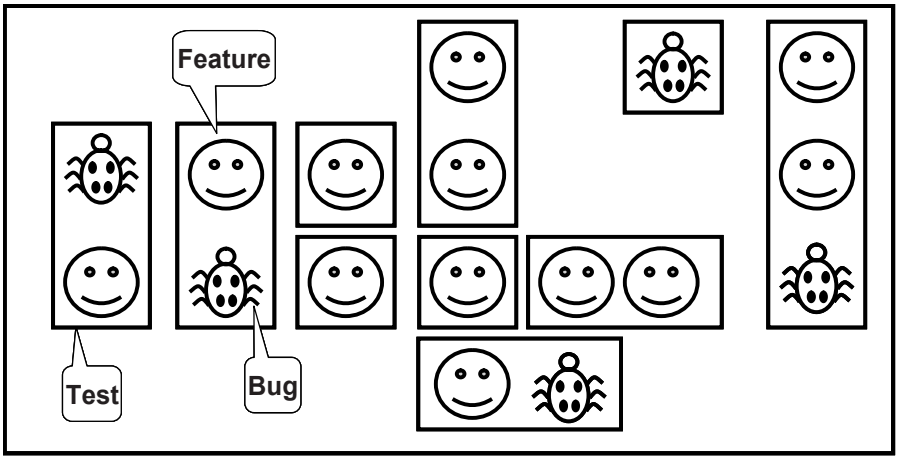


Figure 1-2 shows the total design space and features that are covered by directed testcases. In this space are many features, some of which have bugs. You need to write tests that cover all the features and find the bugs.

1.5 Methodology Basics

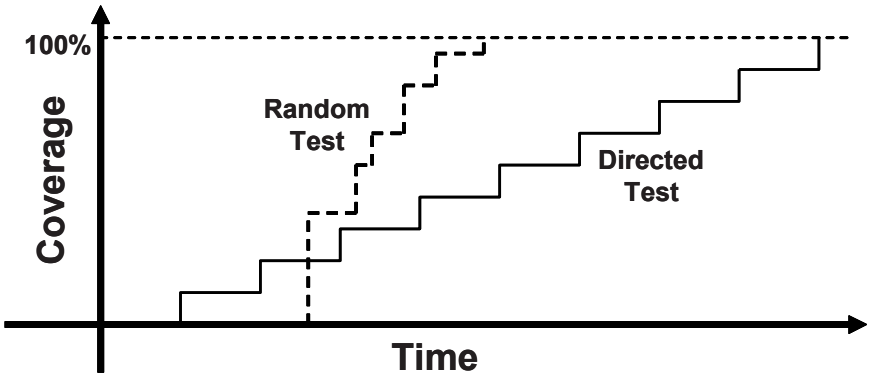
This book uses the following principles.

- Constrained-random stimulus
- Functional coverage
- Layered testbench using transactors
- Common testbench for all tests
- Test-specific code kept separate from testbench

All these principles are related. Random stimulus is crucial for exercising complex designs. A directed test finds the bugs you expect to be in the design, whereas a random test can find bugs you never anticipated. When using random stimulus, you need functional coverage to measure verification progress. Furthermore, once you start using automatically generated stimulus, you need an automated way to predict the results – generally a scoreboard or reference model. Building the testbench infrastructure, including self-prediction, takes a significant amount of work. A layered testbench helps you control the complexity by breaking the problem into manageable pieces. Transactors provide a useful pattern for building these pieces. With appropriate planning, you can build a testbench infrastructure that can be shared by all tests and does not have to be continually modified. You just need to leave "hooks" where the tests can perform certain actions such as shaping the stimulus and injecting disturbances. Conversely, code specific to a single test must be kept separate from the testbench to prevent it from complicating the infrastructure.

Building this style of testbench takes longer than a traditional directed testbench – especially the self-checking portions. As a result, there may be a significant delay before the first test can be run. This gap can cause a manager to panic, and so make this effort part of your schedule. In Figure 1-3, you can see the initial delay before the first random test runs.

Figure 1-3 Constrained-random test progress over time vs. directed testing



While this up-front work may seem daunting, the payback is high. Every random test you create shares this common testbench, as opposed to directed tests where each is written from scratch. Each random test contains a few dozen lines of code to constrain the stimulus in a certain direction and cause any desired exceptions, such as creating a protocol violation. The result is that your single constrained-random testbench is now finding bugs faster than the many directed ones.

As the rate of discovery begins to drop off, you can create new random constraints to explore new areas. The last few bugs may only be found with directed tests, but the vast majority of bugs will be found with random tests.

1.6 Constrained-Random Stimulus

Although you want the simulator to generate the stimulus, you don't want totally random values. You use the SystemVerilog language to describe the format of the stimulus ("address is 32-bits; opcode is ADD, SUB or STORE; length < 32 bytes"), and the simulator picks values that meet the constraints. Constraining the random values to become relevant stimuli is covered in Chap. 6. These values are sent into the design, and are also sent into a high-level model that predicts what the result should be. The design's actual output is compared with the predicted output.

Figure 1-4 shows the coverage for constrained-random tests over the total design space. First, notice that a random test often covers a wider space than a directed one. This extra coverage may overlap other tests, or may explore new areas that you did not anticipate. If these new areas find a bug, you are in luck! If the new area is not legal, you need to write more constraints to keep random generation from creating illegal design functionality. Lastly, you may still have to write a few directed tests to find cases not covered by any other constrained-random tests.

Figure 1-4 Constrained-random test coverage

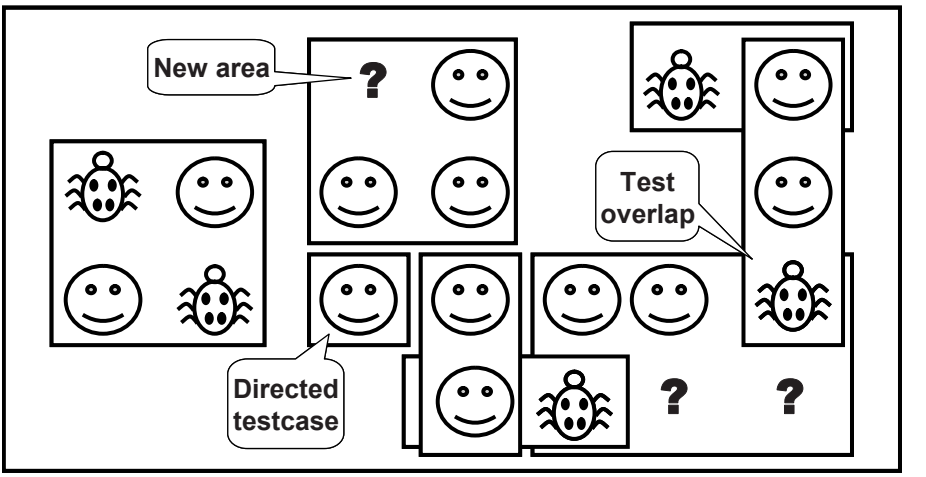
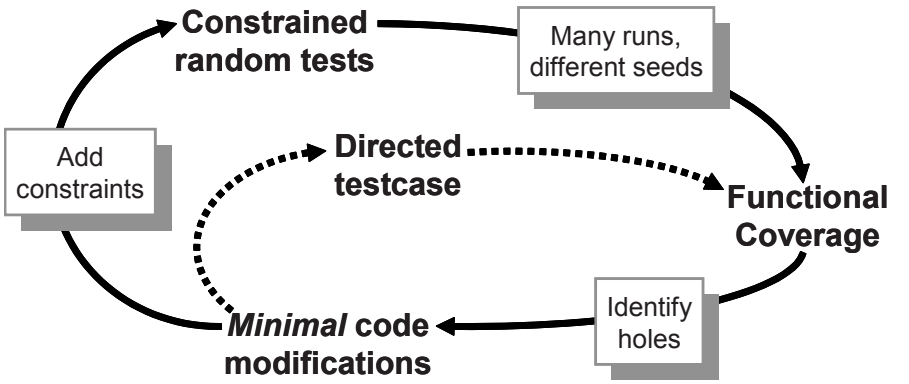


Figure 1-5 shows the paths to achieve complete coverage. Start at the upper left with basic constrained-random tests. Run them with many different seeds. When you look at the functional coverage reports, find the holes where there are gaps in the coverage. Now you make minimal code changes, perhaps by using new constraints, or by injecting errors or delays into the DUT. Spend most of your time in this outer loop, writing directed tests for only the few features that are very unlikely to be reached by random tests.

Figure 1-5 Coverage convergence



1.7 What Should You Randomize?

When you think of randomizing the stimulus to a design, the first thing that you might think of is the data fields. This stimulus is the easiest to create – just call `$random()`. The problem is that this gives a very low payback in terms of bugs found. The primary types of bugs found with random data are data path errors, perhaps with bit-level mistakes. You need to find bugs in the control logic.

You need to think broadly about all design inputs, such as the following.

- Device configuration
- Environment configuration
- Input data
- Protocol exceptions
- Errors and violations
- Delays

These are discussed in Sections. 1.7.1 – 1.7.4.

1.7.1 Device and Environment Configuration

What is the most common reason why bugs are missed during testing of the RTL design? Not enough different configurations are tried. Most tests just use the design as it comes out of reset, or apply a fixed set of initialization vectors to put it into a known state. This is like testing a PC's operating system right after it has been installed, but without any of the applications installed. Of course the performance is fine, and there aren't any crashes.

In a real world environment, the DUT's configuration becomes more random the longer it is in use. For example, I helped a company verify a time-division multiplexor switch that had 2000 input channels and 12 output channels. The verification engineer said, "These channels could be mapped to various configurations on the other side. Each input could be used as a single channel, or further divided into multiple channels. The tricky part is that although a few standard ways of breaking it down are used most of the time, any combination of breakdowns is legal, leaving a huge set of possible customer configurations."

To test this device, the engineer had to write several dozen lines of directed testbench code to configure each channel. As a result, she was never able to try configurations with more than a handful of channels. Together, we wrote a testbench that randomized the parameters for a single channel and then put this in a loop to configure all the switch's channels. Now she had confidence that her tests would uncover configuration-related bugs that would have been missed before.

In the real world, your device operates in an environment containing other components. When you are verifying the DUT, it is connected to a testbench that mimics this environment. You should randomize the entire environment configuration, including the length of the simulation, number of devices, and how they are configured. Of course you need to create constraints to make sure the configuration is legal.

In another Synopsys customer example, a company created an I/O switch chip that connected multiple PCI buses to an internal memory bus. At the start of simulation they randomly chose the number of PCI buses (1–4), the number of devices on each bus (1–8), and the parameters for each device (master or slave, CSR addresses, etc.). They kept track of the tested combinations using functional coverage so that they could be sure that they had covered almost every possible one.

Other environment parameters include test length, error injection rates, and delay modes. See Bergeron (2006) for more examples.

1.7.2 Input Data

When you read about random stimulus, you probably thought of taking a transaction such as a bus write or ATM cell and filling the data fields with random values. Actually, this approach is fairly straightforward as long as you carefully prepare your transaction classes as shown in Chaps. 5 and 8. You need to anticipate any layered protocols and error injection, plus scoreboarding and functional coverage.

1.7.3 Protocol Exceptions, Errors, and Violations

There are few things more frustrating than when a device such as a PC or cell phone locks up. Many times, the only cure is to shut it down and restart. Chances are that deep inside the product there is a piece of logic that experienced some sort of error condition from which it could not recover, and thus prevented the device from working correctly.

How can you prevent this from happening to the hardware you are building? If something can go wrong in the real hardware, you should try to simulate it. Look at all the errors that can occur. What happens if a bus transaction does not complete? If an invalid operation is encountered? Does the design specification state that two signals are mutually exclusive? Drive them both and make sure the device continues to operate properly.

Just as you are trying to provoke the hardware with ill-formed commands, you should also try to catch these occurrences. For example, recall those mutually exclusive signals. You should add checker code to look for these violations. Your code should at least print a warning message when this occurs, and preferably generate an error and wind down the test. It is frustrating to spend hours tracking back through code trying to find the root of a malfunction, especially when you could have caught it close to the source with a simple assertion. (See Vijayaraghavan and Ramanathan ‘2005’ for

more guidelines on writing assertions in your testbench and design code.) Just make sure that you can disable the code that stops simulation on error so that you can easily test error handling.

1.7.4 Delays and Synchronization

How fast should your testbench send in stimulus? Always use constrained-random delays to help catch protocol bugs. A test that uses the shortest delays runs the fastest, but it won't create all possible stimulus. You can create a testbench that talks to another block at the fastest rate, but subtle bugs are often revealed when intermittent delays are introduced.

A block may function correctly for all possible permutations of stimulus from a single interface, but subtle errors may occur when transactions are flowing into multiple inputs. Try to coordinate the various drivers so that they can communicate at different timing rates. What if the inputs arrive at the fastest possible rate, but the output is being throttled back to a slower rate? What if stimulus arrives at multiple inputs concurrently? What if it is staggered with different delays? Use functional coverage, which will be discussed in Chap. 9, to measure what combinations have been randomly generated.

1.7.5 Parallel Random Testing

How should you run the tests? A directed test has a testbench that produces a unique set of stimulus and response vectors. To change the stimulus, you need to change the test. A random test consists of the testbench code plus a random seed. If you run the same test 50 times, each time with a unique seed, you will get 50 different sets of stimuli. Running with multiple seeds broadens the coverage of your test and leverages your work.

You need to choose a unique seed for each simulation. Some people use the time of day, but that can still cause duplicates. What if you are using a batch queuing system across a CPU farm and tell it to start 10 jobs at midnight? Multiple jobs could start at the same time but on different computers, and will thus get the same random seed and run the same stimulus. You should blend in the processor name to the seed. If your CPU farm includes multiprocessor machines, you could have two jobs start running at midnight with the same seed, and so you should also throw in the process ID. Now all jobs get unique seeds.



You need to plan how to organize your files to handle multiple simulations. Each job creates a set of output files, such as log files and functional coverage data. You can run each job in a different directory, or you can try to give a unique name to each file. The easiest approach is to append the random seed value to the directory name.

1.8 Functional Coverage

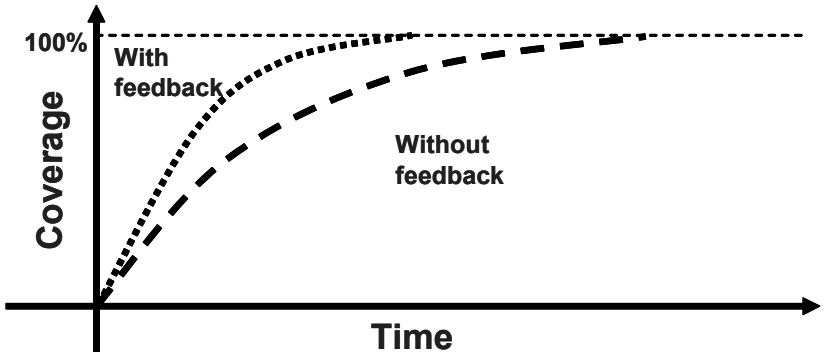
Sections 1.6 and 1.7 showed how to create stimuli that can randomly walk through the entire space of possible inputs. With this approach, your testbench visits some areas often, but takes too long to reach all possible states. Unreachable states will never be visited, even given unlimited simulation time. You need to measure what has been verified in order to check off items in your verification plan.

The process of measuring and using functional coverage consists of several steps. First, you add code to the testbench to monitor the stimulus going into the device, and its reaction and response, to determine what functionality has been exercised. Run several simulations, each with a different seed. Next, merge the results from these simulations into a report. Lastly, you need to analyze the results and determine how to create new stimulus to reach untested conditions and logic. Chapter 9 describes functional coverage in SystemVerilog.

1.8.1 Feedback from Functional Coverage to Stimulus

A random test evolves using feedback. The initial test can be run with many different seeds, thus creating many unique input sequences. Eventually the test, even with a new seed, is less likely to generate stimulus that reaches areas of the design space. As the functional coverage asymptotically approaches its limit, you need to change the test to find new approaches to reach uncovered areas of the design. This is known as "coverage-driven verification" and is shown in Figure 1-6.

Figure 1-6 Test progress with and without feedback



What if your testbench were smart enough to do this for you? In a previous job, I wrote a test that generated every bus transaction for a processor, and additionally fired every bus terminator (Success, Parity error, Retry) in every cycle. This was before HVLs,

and so I wrote a long set of directed tests and spent days lining up the terminator code to fire at just the right cycles. After much hand analysis I declared success – 100% coverage. Then the processor's timing changed slightly! Now I had to reanalyze the test and change the stimuli.

A more productive testing strategy uses random transactions and terminators. The longer you run it, the higher the coverage. As a bonus, the test can be made flexible enough to create valid stimuli even if the design's timing changed. You can accomplish this by adding a feedback loop that looks at the stimulus created so far (generated all write cycles yet?) and then change the constraint weights (drop write weight to zero). This improvement would greatly reduce the time needed to get to full coverage, with little manual intervention.

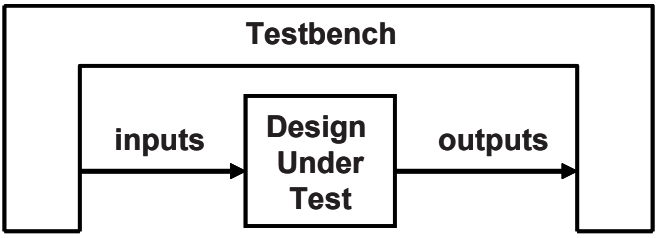
This is not a typical situation, however, because of the trivial feedback from functional coverage to the stimulus. In a real design, how should you change the stimulus to reach a desired design state? This requires deep knowledge of the design and powerful formal techniques. There are no easy answers, and so dynamic feedback is rarely used for constrained-random stimulus. Instead, you need to manually analyze the functional coverage reports and alter your random constraints.

Feedback is used in formal analysis tools such as Magellan (Synopsys, 2003). It analyzes a design to find all the unique, reachable states. It then runs a short simulation to see how many states were visited. Lastly, it searches from the state machine to the design inputs to calculate the stimulus needed to reach any remaining states, and then Magellan applies this to the DUT.

1.9 Testbench Components

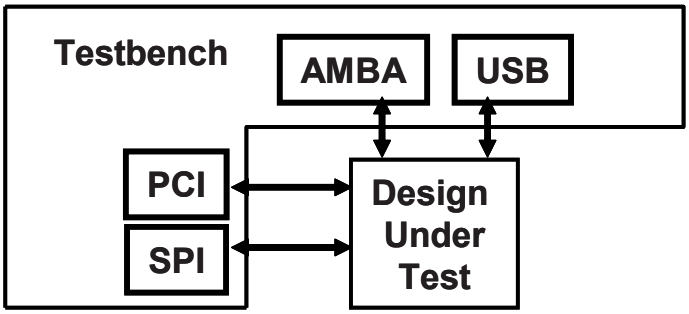
In simulation, the testbench wraps around the DUT, just as a hardware tester connects to a physical chip, as shown in Figure 1-7. Both the testbench and tester provide stimulus and capture responses. The difference between them is that your testbench needs to work over a wide range of levels of abstraction, creating transactions and sequences, which are eventually transformed into bit vectors. A tester just works at the bit level.

Figure 1-7 The testbench – design environment



What goes into that testbench block? It is comprised of many bus functional models (BFM), which you can think of as testbench components – to the DUT they look like real components, but they are part of the testbench, not the RTL design. If the real device connects to AMBA, USB, PCI, and SPI buses, you have to build equivalent components in your testbench that can generate stimulus and check the response, as shown in Figure 1-8. These are not detailed, synthesizable models, but instead high-level transactors that obey the protocol, and execute more quickly. If you are prototyping using FPGAs or emulation, the BFMs do need to be synthesizable.

Figure 1-8 Testbench components



1.10 Layered Testbench

A key concept for any modern verification methodology is the layered testbench. Although this process may seem to make the testbench more complex, it actually helps to make your task easier by dividing the code into smaller pieces that can be developed separately. Don't try to write a single routine that can randomly generate all types of stimulus, both legal and illegal, plus inject errors with a multilayer protocol. The routine quickly becomes complex and unmaintainable.

1.10.1 A Flat Testbench

When you first learned Verilog and started writing tests, they probably looked like the low-level code in Sample 1.1, which does a simplified APB (AMBA Peripheral Bus) Write. (VHDL users may have written similar code.)

Sample 1.1 Driving the APB pins

```
module test(PAddr, PWrite, PSel, PWData, PEnable, Rst, clk);
// Port declarations omitted...

  initial begin
    // Drive reset
    Rst <= 0;
    #100 Rst <= 1;

    // Drive the control bus
    @(posedge clk)
    PAddr  <= 16Õh50;
    PWData <= 32Õh50;
    PWrite <= 1'b1;
    PSel   <= 1'b1;

    // Toggle PEnable
    @(posedge clk)
      PEnable <= 1'b1;
    @(posedge clk)
      PEnable <= 1'b0;

    // Check the result
    if (top.mem.memory[16Õh50] == 32Õh50)
      $display("Success");
    else
      $display("Error, wrong value in memory");
    $finish;
  end
endmodule
```

After a few days of writing code like this, you probably realized that it is very repetitive, and so you created tasks for common operations such as a bus write, as shown in Sample1.2.

Sample 1.2 A task to drive the APB pins

```
task write(reg [15:0] addr, reg [31:0] data);
  // Drive Control bus
  @(posedge clk)
  PAddr  <= addr;
  PWData <= data;
  PWrite <= 1'b1;
  PSel   <= 1'b1;

  // Toggle Penable
  @(posedge clk)
    PEnable <= 1'b1;
  @(posedge clk)
    PEnable <= 1'b0;
endtask
```

Now your testbench became simpler, as shown in Sample 1.3

Sample 1.3 Low-level Verilog test

```
module test(PAddr,PWrite,PSel,PWData,PEnable,Rst,clk);
  // Port declarations omitted...

  // Tasks as shown in Sample 1.2

  initial begin
    reset();                 // Reset the device
    write(16'h50, 32'h50);   // Write data into memory

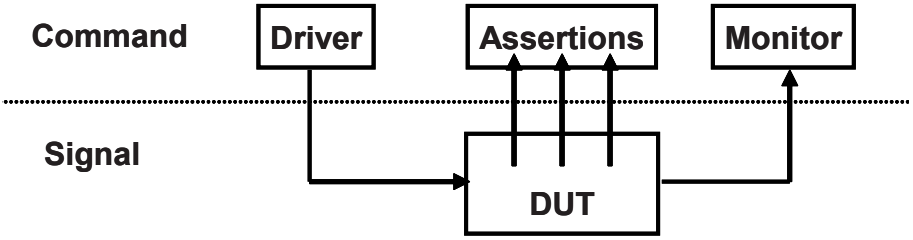
    // Check the result
    if (top.mem.memory[16'h50] == 32'h50)
      $display("Success");
    else
      $display("Error, wrong value in memory");
    $finish;
  end
endmodule
```

By taking the common actions (such as reset, bus reads, and writes) and putting them in a routine, you became more efficient and made fewer mistakes. This creation of the physical and command layers is the first step to a layered testbench.

1.10.2 The Signal and Command Layers

Figure 1-9 shows the lower layers of a testbench.

Figure 1-9 Signal and command layers



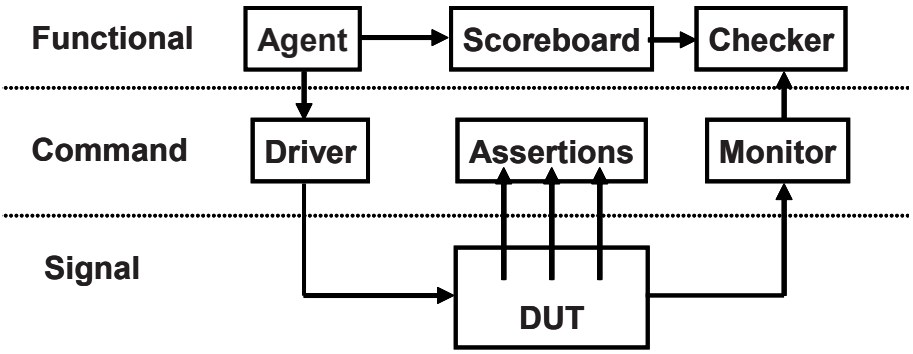
At the bottom is the signal layer that contains the DUT and the signals that connect it to the testbench.

The next higher level is the command layer. The DUT's inputs are driven by the driver that runs single commands, such as bus read or write. The DUT's output drives the monitor that takes signal transitions and groups them together into commands. Assertions also cross the command/signal layer, as they look at individual signals but look for changes across an entire command.

1.10.3 The Functional Layer

Figure 1-10 shows the testbench with the functional layer added, which feeds down into the command layer. The agent block (called the transactor in the VMM) receives higher-level transactions such as DMA read or write and breaks them into individual commands. These commands are also sent to the scoreboard that predicts the results of the transaction. The checker compares the commands from the monitor with those in the scoreboard.

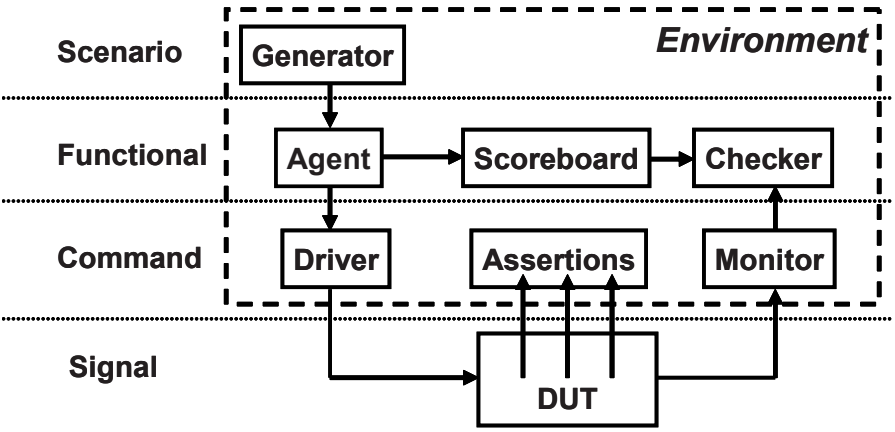
Figure 1-10 Testbench with functional layer added



1.10.4 The Scenario Layer

The functional layer is driven by the generator in the scenario layer, as shown in Figure 1-11. What is a scenario? Remember that your job as a verification engineer is to make sure that this device accomplishes its intended task. An example device is an MP3 player that can concurrently play music from its storage, download new music from a host, and respond to input from the user, such as adjusting the volume and track controls. Each of these operations is a scenario. Downloading a music file takes several steps, such as control register reads and writes to set up the operation, multiple DMA writes to transfer the song, and then another group of reads and writes. The scenario layer of your testbench orchestrates all these steps with constrained-random values for parameters such as track size and memory location.

Figure 1-11 Testbench with scenario layer added



The blocks in the testbench environment (inside the dashed line of Figure 1-11) are written at the beginning of development. During the project they may evolve and you may add functionality, but these blocks should not change for individual tests. This is done by leaving "hooks" in the code so that a test can change the behavior of these blocks without having to rewrite them. You create these hooks with factory patterns (Section 8.2) and callbacks (Section 8.7).

1.10.5 The Test Layer and Functional Coverage

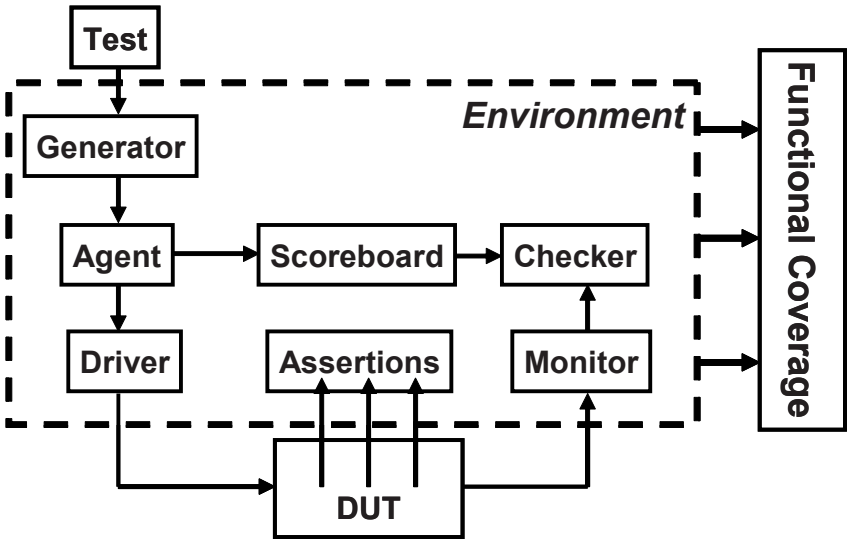
You are now at the top of the testbench, in the test layer, as shown in Figure 1-12. Design bugs that occur between DUT blocks are harder to find as they involve multiple people reading and interpreting multiple specifications.

This top-level test is the conductor: he does not play any musical instrument, but instead guides the efforts of others. The test contains the constraints to create the stimulus.

Functional coverage measures the progress of all tests in fulfilling the verification plan requirements. The functional coverage code changes through the project as the various criteria complete. This code is constantly being modified, and thus it is not part of the environment.

You can create a "directed test" in a constrained-random environment. Simply insert a section of directed test code into the middle of a random sequence, or put the two pieces of code in parallel. The directed code performs the work you want, but the random "background noise" may cause an bug to become visible, perhaps in a block that you never considered.

Figure 1-12 Full testbench with all layers



Do you need all these layers in your testbench? The answer depends on what your DUT looks like. A complicated design requires a sophisticated testbench. You always need the test layer. For a simple design, the scenario layer may be so simple that you can merge it with the agent. When estimating the effort to test a design, don't count the number of gates; count the number of designers. Every time you add another person to the team, you increase the chance of different interpretations of the specifications.

You may need more layers. If your DUT has several protocol layers, each should get its own layer in the testbench environment. For example, if you have TCP traffic that

is wrapped in IP and sent in Ethernet packets, consider using three separate layers for generation and checking. Better yet, use existing verification components.

One last note about Figure 1-12. It shows some of the possible connections between blocks, but your testbench may have a different set. The test may need to reach down to the driver layer to force physical errors. What has been described here is just guidelines – let your needs guide what you create.

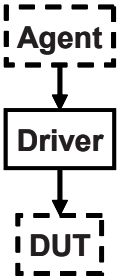
1.11 Building a Layered Testbench

Now it is time to take the preceding figures and learn how to map the components into SystemVerilog constructs.

1.11.1 Creating a Simple Driver

First, take a closer look at one of the blocks, the driver.

Figure 1-13 Connections for the driver



The driver shown in Figure 1-13 receives commands from the agent. The driver may inject errors or add delays. It then breaks down the command into individual signal changes such as bus requests and handshakes. The general term for such a testbench block is a "transactor," which, at its core, is a loop: Sample code for transactor is shown in Sample 1.4.

Sample 1.4 Basic transactor code

```
task run();
  done = 0;
  while (!done) begin
    // Get the next transaction
    // Make transformations
    // Send out transactions
  end
endtask
```

Chapter 5 presents basic OOP and how to create an object that includes the routines and data for a transactor. Another example of a transactor is the agent. It might break apart a complex transaction such as a DMA read into multiple bus commands. Also in Chap. 5, you will see how to build an object that contains the data and routines that make up a command. These objects are sent between transactors using SystemVerilog mailboxes. In Chap. 7, you will learn about many ways to exchange data between the different layers and to synchronize the transactors.

1.12 Simulation Environment Phases

Up until now you have been learning what parts make up the environment. When do these parts execute? You want to clearly define the phases to coordinate the testbench so that all the code for a project works together. The three primary phases are Build, Run, and Wrap-up. Each is divided into smaller steps.

The Build phase is divided into the following steps:

- *Generate configuration*: Randomize the configuration of the DUT and the surrounding environment.
- *Build environment*: Allocate and connect the testbench components based on the configuration. A testbench component is one that only exists in the testbench, as opposed to physical components in the design that are built with RTL code. For example, if the configuration chose three bus drivers, the testbench would allocate and initialize them in this step.
- *Reset the DUT*.
- *Configure the DUT*: Based on the generated configuration from the first step, load the DUT command registers.

The Run phase is where the test actually runs. It has the following steps:

- *Start environment*: Run the testbench components such as BFMs and stimulus generators.
- *Run the test*: Start the test and then wait for it to complete. It is easy to tell when a directed test has completed, but doing so can be complex for a random test. You can use the testbench layers as a guide. Starting from the top, wait for a layer to drain all the inputs from the previous layer (if any), wait for the current layer to become idle, and then wait for the next lower layer. You should also use time-out checkers to ensure that the DUT or testbench does not lock up.

The Wrap-up phase has two steps:

- *Sweep*: After the lowest layer completes, you need to wait for the final transactions to drain out of the DUT.
- *Report*: Once the DUT is idle, sweep the testbench for lost data. Sometimes the scoreboard holds transactions that never came out, perhaps because they were dropped by the DUT. Armed with this information, you can create the final report on whether the test passed or failed. If it failed, be sure to delete any functional coverage results, as they may not be correct.

As shown in Figure 1-12, the test starts the environment, which, in turn, runs each of the steps. More details can be found in Chap. 8.

1.13 Maximum Code Reuse

To verify a complex device with hundreds of features, you have to write hundreds of directed tests. If you use constrained-random stimulus, you will write far fewer tests. Instead, the real work is put into constructing the testbench, which contains all the lower testbench layers: scenario, functional, command, and signal. This testbench code is used by all the tests, and so it remains generic.

These guidelines appear to recommend an overly complicated testbench, but remember that every line that you put into a testbench can eliminate a line in every single test. If you know you will be creating a few dozen tests, there is a high payback in making a more sophisticated testbench. Keep this in mind when you read Chap. 8.

1.14 Testbench Performance

If this is the first time you have seen this methodology, you probably have some qualms about how it works compared to directed testing. A common objection is testbench performance. A directed test often simulates in less than a second, whereas constrained-random tests will wander around through the state space for minutes or even hours. The problem with this argument is that it ignores a real verification bottleneck: the time required by you to create a test. You may be able to hand-craft a directed test in a day, and debug it and manually verify the results by hand in another day or two. The actual simulation run-time is dwarfed by the amount of time that you personally invested.

There are several steps to creating a constrained-random test. The first and most significant step is building the layered testbench, including the self-checking portion. The benefit of this work is shared by all tests, and so it is well worth the effort. The second step is creating the stimulus specific to a goal in the verification plan. You

may be crafting random constraints, or devious ways of injecting errors or protocol violations. Building one of these may take more time than making several directed tests, but the payoff will be much higher. A constrained-random test that tries thousands of different protocol variations is worth more than the handful of directed tests that could have been created in the same amount of time.

The third step in constrained-random testing is functional coverage. This task starts with the creation of a strong verification plan with clear goals that can be easily measured. Next you need to create the SystemVerilog code that adds instrumentation to the environment and gathers the data. Finally, it is essential that you need to analyze the results to determine if you have met the goals, and if not, how you should modify the tests.

1.15 Conclusion

The continuous growth in complexity of electronic designs requires a modern, systematic, and automated approach to creating testbenches. The cost of fixing a bug grows by tenfold as a project moves from each step of specification to RTL coding, gate synthesis, fabrication, and finally into the user's hands. Directed tests only test one feature at a time and cannot create the complex stimulus and configurations that the device would be subjected to in the real world. To produce robust designs, you must use constrained-random stimulus combined with functional coverage to create the widest possible range of stimulus.

Chapter 2

Data Types

SystemVerilog offers many improved data structures compared with Verilog. Some of these were created for designers but are also useful for testbenches. In this chapter, you will learn about the data structures most useful for verification.

SystemVerilog introduces new data types with the following benefits.

- Two-state: better performance, reduced memory usage
- Queues, dynamic and associative arrays: reduced memory usage, built-in support for searching and sorting
- Classes and structures: support for abstract data structures
- Unions and packed structures: allow multiple views of the same data
- Strings: built-in string support
- Enumerated types: code is easier to write and understand

2.1 Built-In Data Types

Verilog-1995 has two basic data types: variables and nets, both which hold 4-state values: 0, 1, Z, and X. RTL code uses variables to store combinational and sequential values. Variables can be unsigned single or multi-bit (`reg [7:0] m`), signed 32-bit variables (`integer`), unsigned 64-bit variables (`time`), and floating point numbers (`real`). Variables can be grouped together into arrays that have a fixed size. All storage is static, meaning that all variables are alive for the entire simulation and routines cannot use a stack to hold arguments and local values. A net is used to connect parts

of a design such as gate primitives and module instances. Nets come in many flavors, but most designers use scalar and vector wires to connect together the ports of design blocks.

SystemVerilog adds many new data types to help both hardware designers and verification engineers.

2.1.1 The Logic Type

The one thing in Verilog that always leaves new users scratching their heads is the difference between a `reg` and a `wire`. When driving a port, which should you use? How about when you are connecting blocks? SystemVerilog improves the classic `reg` data type so that it can be driven by continuous assignments, gates, and modules, in addition to being a variable. It is given the synonym `logic` so that it does not look like a register declaration. A `logic` signal can be used anywhere a net is used, except that a `logic` variable cannot be driven by multiple structural drivers, such as when you are modeling a bidirectional bus. In this case, the variable needs to be a net-type such as `wire` so that SystemVerilog can resolve the multiple values to determine the final value.

Sample 2.1 shows the SystemVerilog `logic` type.

Sample 2.1 Using the logic type

```
module logic_data_type(input logic rst_h);
  parameter CYCLE = 20;
  logic q, q_l, d, clk, rst_l;
  initial begin
    clk = 0;                    // Procedural assignment
    forever #(CYCLE/2) clk = ~clk;
  end

  assign rst_l = ~rst_h;        // Continuous assignment
  not n1(q_l, q);               // q_l is driven by gate
  my_dff d1(q, d, clk, rst_l); // q is driven by module

endmodule
```



You can use the `logic` type to find netlist bugs as this type can only have a single driver. Rather than trying to choose between `reg` and `wire`, declare all your signals as `logic`, and you'll get a compilation error if it has multiple drivers. Of course, any signal that you do want to have multiple drivers, such as a bidirectional bus, should be declared with a net type such as `wire`.

2.1.2 2-State Data Types

SystemVerilog introduces several 2-state data types to improve simulator performance and reduce memory usage, compared with variables declared as 4-state types. The simplest type is the `bit`, which is always unsigned. There are four signed 2-state types: `byte`, `shortint`, `int`, and `longint` as shown in Sample 2.2.

Sample 2.2 Signed data types

```
bit b;            // 2-state, single-bit
bit [31:0] b32;   // 2-state, 32-bit unsigned integer
int unsigned ui; // 2-state, 32-bit unsigned integer
int i;            // 2-state, 32-bit signed integer
byte b8;          // 2-state, 8-bit signed integer
shortint s;       // 2-state, 16-bit signed integer
longint l;        // 2-state, 64-bit signed integer
integer i4;       // 4-state, 32-bit signed integer
time t;           // 4-state, 64-bit unsigned integer
real r;           // 2-state, double precision floating pt
```



You might be tempted to use types such as `byte` to replace more verbose declarations such as `logic [7:0]`. Hardware designers should be careful as these new types are signed variables, and so a `byte` variable can only count up to 127, not the 255 you may expect. (It has the range –128 to +127.) You could use `byte unsigned`, but that is more verbose than just `bit [7:0]`. Signed variables can also cause unexpected results with randomization, as discussed in Chap. 6.



Be careful connecting 2-state variables to the design under test, especially its outputs. If the hardware tries to drive an X or Z, these values are converted to a 2-state value, and your testbench code may never know. Don't try to remember if they are converted to 0 or 1; instead, always check for propagation of unknown values. Use the `$isunknown()` operator that returns 1 if any bit of the expression is X or Z, as shown in Sample 2.3.

Sample 2.3 Checking for 4-state values

```
if ($isunknown(iport) == 1)
  $display("@%0t: 4-state value detected on iport %b",
           $time, iport);
```

The format `%0t` and the argument `$time` print the current simulation time, formatted as specified with the `$timeformat`() routine. Time values are explored in more detail in Section 3.7.

2.2 Fixed-Size Arrays

SystemVerilog offers several flavors of arrays beyond the single-dimension, fixed-size Verilog-1995 arrays. Many enhancements have been made to these classic arrays.

2.2.1 Declaring and Initializing Fixed-Size Arrays

Verilog requires that the low and high array limits must be given in the declaration. Since almost all arrays use a low index of 0, SystemVerilog lets you use the shortcut of just giving the array size, which is similar to C's style.

Sample 2.4 Declaring fixed-size arrays

```
int lo_hi[0:15];         // 16 ints [0]..[15]
int c_style[16];         // 16 ints [0]..[15]
```

You can create multidimensional fixed-size arrays by specifying the dimensions after the variable name. Sample 2.5 creates several two-dimensional arrays of integers, 8 entries by 4, and sets the last entry to 1. Multidimensional arrays were introduced in Verilog-2001, but the compact declaration style is new.

Sample 2.5 Declaring and using multidimensional arrays

```
int array2 [0:7][0:3];  // Verbose declaration
int array3 [8][4];      // Compact declaration
array2[7][3] = 1;       // Set last array element
```

If your code accidently tries to read from an out-of-bounds address, SystemVerilog will return the default value for the array element type. That just means that an array of 4-state types, such as `logic`, will return X's, whereas an array of 2-state types, such as `int` or `bit`, will return 0. This applies for all array types – fixed, dynamic, associative, or queue, and also if your address has an X or Z. An undriven net is Z.

Many SystemVerilog simulators store each element on a 32-bit word boundary. So a `byte`, `shortint`, and `int` are all stored in a single word, whereas a `longint` is stored in two words.

An unpacked array, such as the one shown in Sample 2.6, stores the values in the lower portion of the word, whereas the upper bits are unused. The array of bytes, `b_unpack`, is stored in three words, as shown in Figure 2-1.

Sample 2.6 Unpacked array declarations

```
bit [7:0] b_unpack[3];   // Unpacked
```

Figure 2-1 Unpacked array storage



Packed arrays are explained in Section 2.2.6.

Simulators generally store 4-state types such as `logic` and `integer` in two or more consecutive words, using twice the storage as 2-state variables.

2.2.2 The Array Literal

Sample 2.7 shows how to initialize an array using an array literal, which is an apostrophe followed by the values in curly braces. (This is not the accent grave used for compiler directives and macros.) You can set some or all elements at once. You are able to replicate values by putting a count before the curly braces. Lastly, you might specify a default value for any element that does not have an explicit value.

Sample 2.7 Initializing an array

```
int ascend[4] = Õ{0,1,2,3}; // Initialize 4 elements
int descend[5];

descend = Õ{4,3,2,1,0};         // Set 5 elements
descend[0:2] = Õ{5,6,7};        // Set first 3 elements
ascend = Õ{4{8}};               // Four values of 8
descend = Õ{9, 8, default:-1}; // {9, 8, -1, -1, -1}
```

2.2.3 Basic Array Operations – `for` and `foreach`

The most common way to manipulate an array is with a `for`- or `foreach`-loop. In Sample 2.8, the variable `i` is declared local to the `for`-loop. The SystemVerilog function `$size` returns the size of the array. In the `foreach`-loop, you specify the array name and an index in square brackets, and SystemVerilog automatically steps through all the elements of the array. The index variable is automatically declared for you and is local to the loop.

Sample 2.8 Using arrays with `for-` and `foreach`-loops

```
initial begin
  bit [31:0] src[5], dst[5];
  for (int i=0; i<$size(src); i++)
    src[i] = i;
  foreach (dst[j])
    dst[j] = src[j] * 2;  // dst doubles src values
end
```

Note that in Sample 2.9, the syntax of the `foreach`-loop for multidimensional arrays may not be what you expected! Instead of listing each subscript in separate square brackets – `[i][j]` – they are combined with a comma – `[i,j]`.

Sample 2.9 Initialize and step through a multidimensional array

```
int md[2][3] = õ{õ{0,1,2}, õ{3,4,5}};
initial begin
  $display("Initial value:");
  foreach (md[i,j])   // Yes, this is the right syntax
    $display("md[%0d][%0d] = %0d", i, j, md[i][j]);

  $display("New value:");
  // Replicate last 3 values of 5
  md = õ{õ{9, 8, 7}, õ{3{õ5}}};
  foreach (md[i,j])   // Yes, this is the right syntax
    $display("md[%0d][%0d] = %0d", i, j, md[i][j]);
end
```

The output from Sample 2.9 is shown in Sample 2.10.

Sample 2.10 Output from printing multidimensional array values

```
Initial value:
md[0][0] = 0
md[0][1] = 1
md[0][2] = 2
md[1][0] = 3
md[1][1] = 4
md[1][2] = 5
New value:
md[0][0] = 9
md[0][1] = 8
md[0][2] = 7
md[1][0] = 5
md[1][1] = 5
md[1][2] = 5
```

You can omit some dimensions in the `foreach`-loop if you don't need to step through all of them. Sample 2.11 prints a two-dimensional array in a rectangle. It

steps through the first dimension in the outer loop, and then through the second dimension in the inner loop.

Sample 2.11 Printing a multidimensional array

```
initial begin
  byte twoD[4][6];
  foreach(twoD[i,j])
    twoD[i][j] = i*10+j;

  foreach (twoD[i]) begin    // Step through first dim.
    $write("%2d:", i);
    foreach(twoD[,j])        // Step through second
      $write("%3d", twoD[i][j]);
    $display;
  end
end
```

Sample 2.11 produces the following output.

Sample 2.12 Output from printing multidimensional array values

```
 0:  0  1  2  3  4  5
 1: 10 11 12 13 14 15
 2: 20 21 22 23 24 25
 3: 30 31 32 33 34 35
```

Lastly, a `foreach`-loop iterates using the ranges in the original declaration. The array `f[5]` is equivalent to `f[0:4]`, and a `foreach(f[i])` is equivalent to `for(int i=0;i<=4; i++)`. With the array `rev[6:2]`, the statement `foreach(rev[i])` is equivalent to `for(int i=6; i>=2; i--)`.

2.2.4 Basic Array Operations – Copy and Compare

You can perform aggregate compare and copy of arrays without loops. (An aggregate operation works on the entire array as opposed to working on just an individual element.) Comparisons are limited to just equality and inequality. Sample 2.13 shows several examples of compares. The `? :` conditional operator is a mini `if`-statement. In Sample 2.13, it is used to choose between two strings. The final compare uses an array slice, `src[1:4]`, which creates a temporary array with four elements.

Sample 2.13 Array copy and compare operations

```
initial begin
  bit [31:0] src[5] = Õ{0,1,2,3,4},
             dst[5] = Õ{5,4,3,2,1};

  // Aggregate compare the two arrays
  if (src==dst)
    $display("src == dst");
  else
    $display("src != dst");

  // Aggregate copy all src values to dst
  dst = src;

  // Change just one element
  src[0] = 5;

  // Are all values equal (no!)
  $display("src %s dst", (src == dst) ? "==" : "!=");

  // Use array slice to compare elements 1-4
  // $display("src[1:4] %s dst [1:4]",
               src[1:4] == dst[1:4]) ? "==" : "!=");
end
```

You cannot perform aggregate arithmetic operations such as addition on arrays. Instead, you can use loops. For logical operations such as `xor`, you have to either use a loop or use packed arrays as described in Section 2.2.6.

2.2.5 Bit and Array Subscripts, Together at Last

A common annoyance in Verilog-1995 is that you cannot use array and bit subscripts together. Verilog-2001 removes this restriction for fixed-size arrays. Sample 2.14 prints the first array element (binary 101), its lowest bit (1), and the next two higher bits (binary 10).

Sample 2.14 Using word and bit subscripts together

```
initial begin
  bit [31:0] src[5] = Õ{5{5}};
  $displayb(src[0],,          // Õb101 or Õd5
            src[0][0],,       // Õb1
            src[0][2:1]);     // Õb10
end
```

Although this change is not new to SystemVerilog, many users may not know about this useful improvement in Verilog-2001.

2.2.6 Packed Arrays

For some data types, you may want both to access the entire value and also to divide it into smaller elements. For example, you may have a 32-bit register that sometimes you want to treat as four 8-bit values and at other times as a single, unsigned value. A SystemVerilog packed array is treated as both an array and a single value. It is stored as a contiguous set of bits with no unused space, unlike an unpacked array.

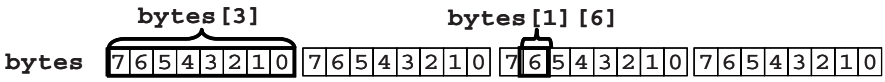
2.2.7 Packed Array Examples

The packed bit and array dimensions are specified as part of the type, before the variable name. These dimensions must be specified in the `[msb:lsb]` format, not `[size]`. Sample 2.15 shows the variable `bytes`, a packed array of four bytes that are stored in a single 32-bit word as shown in Figure 2-2.

Sample 2.15 Packed array declaration and usage

```
bit [3:0] [7:0] bytes;  // 4 bytes packed into 32-bits
bytes = 32ÕhCafe_Dada;
$displayh(bytes,,         // Show all 32-bits
          bytes[3],,      // Most significant byte "CA"
          bytes[3][7]);  // Most significant bit  "1"
```

Figure 2-2 Packed array layout



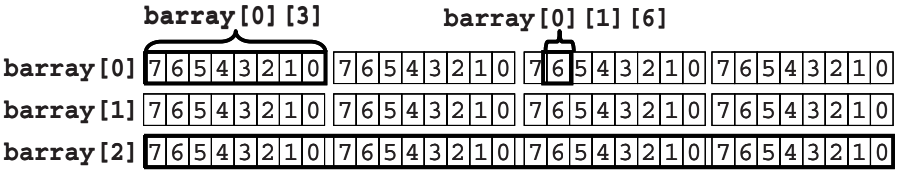
You can mix packed and unpacked dimensions. You may want to make an array that represents a memory that can be accessed as bits, bytes, or longwords. In Sample 2.16, `barray` is an unpacked array of three packed elements.

Sample 2.16 Declaration for a mixed packed/unpacked array

```
bit [3:0] [7:0] barray [3];     // Packed: 3x32-bit
bit [31:0] lw = 32Õh0123_4567; // Word
bit [7:0] [3:0] nibbles;        // Packed array of nibbles
barray[0] = lw;
barray[0][3] = 8Õh01;
barray[0][1][6] = 1Õb1;
nibbles = barray[0];            // Copy packed values
```

The variable `bytes` in Sample 2.15 is a packed array of four bytes that are stored in a single word. `barray` is an array of three of these elements, which are stored in memory as shown in Figure 2-3.

Figure 2-3 Packed array bit layout



With a single subscript, you get a word of data, `barray[2]`. With two subscripts, you get a byte of data, `barray[0][3]`. With three subscripts, you can access a single bit, `barray[0][1][6]`. Note that because one dimension is specified after the name, `barray[3]`, that dimension is unpacked, and so you always need to use at least one subscript.

The last line of Sample 2.16 copies between two packed arrays. Since the underlying values are just bits, you can copy even if the arrays have different dimensions.

2.2.8 Choosing Between Packed and Unpacked Arrays

Which should you choose – a packed or an unpacked array? A packed array is handy if you need to convert to and from scalars. For example, you might need to reference a memory as a byte or as a word. The `barray` in Figure 2-3 can handle this requirement. Any array type can be packed, including dynamic arrays, queues, and associative arrays, which are explained in Sections 2.3–2.5.

If you need to wait for a change in an array, you have to use a packed array. Perhaps your testbench might need to wake up when a memory changes value, and so you want to use the `@` operator. This is however only legal with scalar values and packed arrays. In Sample 2.16 you can block on the variable `lw`, and `barray[0]`, but not the entire array `barray` unless you expand it: `@(barray[0] or barray[1] or barray[2])`.

2.3 Dynamic Arrays

The basic Verilog array type shown so far is known as a fixed-size array, as its size is set at compile time. What if you do not know the size of the array until run-time? For example, you may want to generate a random number of transactions at the start of simulation. If you stored the transactions in an fixed-size array, it would have to be large enough to hold the maximum number of transactions, but would typically hold far fewer, thus wasting memory. SystemVerilog provides a dynamic array that can be allocated and resized during simulation and so your simulation consumes a minimal amount of memory.

A dynamic array is declared with empty word subscripts `[]`. This means that you do not specify the array size at compile time; instead, you give it at run-time. The array is

initially empty, and so you must call the `new[]` constructor to allocate space, passing in the number of entries in the square brackets. If you pass the name of an array to the `new[]` constructor, the values are copied into the new elements, as shown in Sample 2.17.

Sample 2.17 Using dynamic arrays

```
int dyn[], d2[];                  // Declare dynamic arrays

initial begin
  dyn = new[5];                   // A: Allocate 5 elements
  foreach (dyn[j]) dyn[j] = j;    // B: Initialize the elements
  d2 = dyn;                       // C: Copy a dynamic array
  d2[0] = 5;                      // D: Modify the copy
  $display(dyn[0],d2[0]);         // E: See both values (0 & 5)
  dyn = new[20](dyn);             // F: Allocate 20 ints & copy
  dyn = new[100];                 // G: Allocate 100 new ints
                                  //    Old values are lost
  dyn.delete();                   // H: Delete all elements
end
```

In Sample 2.17, Line A calls `new[5]` to allocate 5 array elements. The dynamic array `dyn` now holds 5 `int`'s. B sets the value of each element of the array to its index value. Line C allocates another array and copies the contents of `dyn` into it. Lines D and E show that the arrays `dyn` and `d2` are separate. Line E allocates 20 new elements, and copies the existing 5 elements of `dyn` to the beginning of the array. Then the old 5-element array is deallocated. The result is that `dyn` points to a 20-element array. The last call to `new[]` allocates 100 elements, but the existing values are not copied. The old 20-element array is deallocated. Finally, line H deletes the `dyn` array.

The `$size` function returns the size of an array. Dynamic arrays have several built-in routines, such as `delete` and `size`.

If you want to declare a constant array of values but do not want to bother counting the number of elements, use a dynamic array with an array literal. In Sample 2.18, there are 9 masks for 8 bits, but you should let SystemVerilog count them, rather than making a fixed-size array and accidently choosing the wrong size of 8.

Sample 2.18 Using a dynamic array for an uncounted list

```
bit [7:0] mask[] = Õ{8Õb0000_0000, 8Õb0000_0001,
                     8Õb0000_0011, 8Õb0000_0111,
                     8Õb0000_1111, 8Õb0001_1111,
                     8Õb0011_1111, 8Õb0111_1111,
                     8Õb1111_1111};
```

You can make assignments between fixed-size and dynamic arrays as long as they have the same base type such as `int`. You can assign a dynamic array to a fixed array as long as they have the same number of elements.

When you copy a fixed-size array to a dynamic array, SystemVerilog calls the `new[]` constructor to allocate space, and then copies the values.

2.4 Queues

SystemVerilog introduces a new data type, the queue, which combines the best of an linked list and array. Like a linked list, you can add or remove elements anywhere in a queue, without the performance hit of a dynamic array that has to allocate a new array and copy the entire contents. Like an array, you can directly access any element with an index, without linked list's overhead of stepping through the preceding elements.

A queue is declared with word subscripts containing a dollar sign: `[$]`. The elements of a queue are numbered from 0 to $. Sample 2.19 shows how you can add and remove values from a queue using methods. Note that queue literals only have curly braces, and are missing the initial apostrophe of array literals.

The SystemVerilog queue is similar to the Standard Template Library's deque data type. You create a queue by adding elements. SystemVerilog typically allocates extra space so that you can quickly insert additional elements. If you add enough elements that the queue runs out of that extra space, SystemVerilog automatically allocates more. As a result, you can grow and shrink a queue without the performance penalty of a dynamic array, and SystemVerilog keeps track of the free space for you. Note that you never call the `new[]` constructor for a queue.

Sample 2.19 Queue operations

```
int j = 1,
    q2[$] = {3,4},          // Queue literals do not use Õ
    q[$] = {0,2,5};         // {0,2,5}

initial begin
  q.insert(1, j);           // {0,1,2,5}     Insert 1 before 2
  q.insert(3, q2);          // {0,1,2,3,4,5} Insert queue in q[1]
  q.delete(1);              // {0,2,3,4,5}   Delete elem. #1

  // These operations are fast
  q.push_front(6);          // {6,0,2,3,4,5} Insert at front
  j = q.pop_back;           // {6,0,2,3,4}   j = 5
  q.push_back(8);           // {6,0,2,3,4,8} Insert at back
  j = q.pop_front;          // {0,2,3,4,8}   j = 6
  foreach (q[i])
    $display(q[i]);         //               Print entire queue
  q.delete();               // {}            Delete entire queue
end
```

You can use word subscripts and concatenation instead of methods. As a shortcut, if you put a $ on the left side of a range, such as `[$:2]`, the `$` stands for the minimum value, `[0:2]`. A `$` on the right side, as in `[1:$]`, stands for the maximum value, `[1:2]`, in first line of the initial block of Sample 2.20.

Sample 2.20 Queue operations

```
int j = 1,
    q2[$] = {3,4},          // Queue literals do not use Õ
    q[$] = {0,2,5};         // {0,2,5}

initial begin               // Result
  q = {q[0], j, q[1:$]};    // {0,1,2,5}     Insert 1 before 2
  q = {q[0:2], q2, q[3:$]}; // {0,1,2,3,4,5} Insert queue in q
  q = {q[0], q[2:$]};       // {0,2,3,4,5}   Delete elem. #1

  // These operations are fast
  q = {6, q};               // {6,0,2,3,4,5} Insert at front
  j = q[$];                 // j = 5         Equivalent of
  q = q[0:$-1];             // {6,0,2,3,4}      pop_back
  q = {q, 8};               // {6,0,2,3,4,8} Insert at back
  j = q[0];                 // j = 6         Equivalent of
  q = q[1:$];               // {0,2,3,4,8}      pop_front

  q = {};                   // {}            Delete entire queue
end
```

[1]Not all SystemVerilog simulators support inserting a queue with the insert() method.

The queue elements are stored in contiguous locations, and so it is efficient to push and pop elements from the front and back. This takes a fixed amount of time no matter how large the queue. Adding and deleting elements in the middle of a queue requires shifting the existing data to make room. The time to do this grows linearly with the size of the queue.

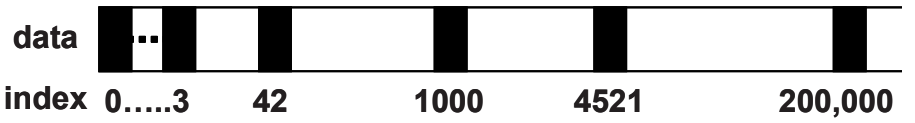
You can copy the contents of a fixed or dynamic array into a queue.

2.5 Associative Arrays

Dynamic arrays are good if you want to occasionally create a big array, but what if you want something really large? Perhaps you are modeling a processor that has a multi-gigabyte address range. During a typical test, the processor may only touch a few hundred or thousand memory locations containing executable code and data, so allocating and initializing gigabytes of storage is wasteful.

SystemVerilog offers associative arrays that store entries in a sparse matrix. This means that while you can address a very large address space, SystemVerilog only allocates memory for an element when you write to it. In the following picture, the associative array holds the values 0:3, 42, 1,000, 4,521, and 200,000. The memory used to store these is far less than would be needed to store a fixed or dynamic array with 200,000 entries.

Figure 2-4 Associative array



An associative array can be stored by the simulator as a tree or hash table. This additional overhead is acceptable when you need to store arrays with widely separated index values, such as packets indexed with 32-bit addresses or 64-bit data values.

An associative array is declared with a data type[2] in square brackets, such as `[int]` or `[Packet]`. Sample 2.21 shows declaring, initializing, and stepping through an associative array.

[2]You can also declare an associative array with wildcard subscripts, as in wild[*]. However, this style is not recommended as you are allowing subscripts of any data type. One of the many problems is with foreach-loops – what type is the variable j in foreach(wild[j])?

Sample 2.21 Declaring, initializing, and using associative arrays

```
initial begin
  bit [63:0] assoc[int], idx = 1;

  // Initialize widely scattered values
  repeat (64) begin
    assoc[idx] = idx;
    idx = idx << 1;
  end

  // Step through all index values with foreach
  foreach (assoc[i])
    $display("assoc[%h] = %h", i, assoc[i]);

  // Step through all index values with functions
  if (assoc.first(idx))
    begin                    // Get first index
    do
      $display("assoc[%h]=%h", idx, assoc[idx]);
    while (assoc.next(idx));  // Get next index
  end

  // Find and delete the first element
  assoc.first(idx);
  assoc.delete(idx);
  $display("The array now has %0d elements", assoc.num);
end
```

Sample 2.21 has the associative array, `assoc`, with very scattered elements: 1, 2, 4, 8, 16, etc. A simple `for`-loop cannot step through them; you need to use a `foreach`-loop. If you want finer control, you can use the `first` and `next` functions in a `do...while` loop. These functions modify the index argument, and return 0 or 1 depending on whether any elements are left in the array.

Associative arrays can also be addressed with a string index, similar to Perl's hash arrays. Sample 2.22 reads a file with strings and builds the associative array `switch` so that you can quickly map from a string value to a number. Strings are explained in more detail in Section 2.14. You can use the function `exists()` to check if an element exists, as shown in Sample 2.22. If you try to read an element that has not been written, SystemVerilog returns the default value for the array type, such as 0 for 2-state types, or X for 4-state types.

Sample 2.22 Using an associative array with a string index

```
/*
Input file contains:
  42   min_address
  1492 max_address
*/

int switch[string], min_address, max_address;
initial begin
  int i, r, file;
  string s;
  file = $fopen("switch.txt", "r");
  while (! $feof(file)) begin
    r = $fscanf(file, "%d %s", i, s);
    switch[s] = i;
  end
  $fclose(file);

  // Get the min address, default is 0
  min_address = switch["min_address"];

  // Get the max address, default = 1000
  if (switch.exists("max_address"))
    max_address = switch["max_address"];
  else
    max_address = 1000;

  // Print all switches
  foreach (switch[s])
    $display("switch[Õ%sÕ]=%0d", s, switch[s]);
end
```

2.6 Linked Lists

SystemVerilog provides a linked list data-structure that is analogous to the STL (Standard Template Library) List container. The container is defined as a parameterized class, meaning that it can be customized to hold data of any type.

Now that you know there is a linked list in SystemVerilog, avoid using it. C++ programmers might be familiar with the STL version, but SystemVerilog's queues are more efficient and easier to use.

2.7 Array Methods

There are many array methods that you can use on any unpacked array types: fixed, dynamic, queue, and associative. These routines can be as simple as giving the current array size or as complex as sorting the elements. The parentheses are optional if there are no arguments.

2.7.1 Array Reduction Methods

A basic array reduction method takes an array and reduces it to a single value, as shown in Sample 2.23. The most common reduction method is `sum`, which adds together all the values in an array. Be careful of SystemVerilog's rules for handling the width of operations. By default, if you add the values of a single-bit array, the result is a single bit. However, if you use it in a 32-bit expression, store the result in a 32-bit variable, compare it to a 32-bit variable, or use the proper `with` expression. SystemVerilog uses 32-bits when adding up the values. The `with` expression is described in Section 2.7.2.

Sample 2.23 Creating the sum of an array

```
bit on[10];  // Array of single bits
int total;

initial begin
  foreach (on[i])
    on[i] = i;   // on[i] gets 0 or 1

  // Print the single-bit sum
  $display("on.sum = %0d", on.sum); // on.sum = 1

  // Print the sum using 32-bit total
  $display("on.sum = %0d", on.sum + 32Õd0); // on.sum = 5

  // Sum the values using 32-bits as total is 32-bits
  total = on.sum;
  $display("total = %0d", total);     // total = 5

  // Compare the sum to a 32-bit value
  if (on.sum >=32Õd5)               // True
    $display("sum has 5 or more 1Õs");

  // Compute with 32-bit signed arithimetic
  $display("int sum=%0d", on.sum with (intÕ(item)));
  end
```

Other array reduction methods are `product`, `and`, `or`, and `xor`.

SystemVerilog does not have a method specifically for choosing a random element from an array, and so use the index `$urandom_range(array.size()-1)` for queues and dynamic arrays, and `$urandom_range($size(array)-1)` for fixed arrays, queues, dynamic, and associative arrays. See Section 6.10 for more information on `$urandom_range`.

If you need to choose a random element from an associative array, you need to step through the elements one by one as there is no direct way to access the *N*th element. Sample 2.24 shows how to pick a random element from an associative array indexed by integers. If the array was indexed by a string, just change the type of `idx` to `string`.

Sample 2.24 Picking a random element from an associative array

```
int aa[int], rand_idx, element, count;

element = $urandom_range(aa.size()-1);
foreach(aa[i])
  if (count++ == element) begin
    rand_idx = i;   // Save the associative array index
    break;          //    and quit
  end

$display("%0d element aa[%0d] = %0d",
         element, rand_idx, aa[rand_idx]);
```

2.7.2 Array Locator Methods

What is the largest value in an array? Does an array contain a certain value? The array locator methods find data in an unpacked array. These methods always return a queue.

Sample 2.25 uses a fixed-size array, `f[6]`, a dynamic array, `d[]`, and a queue, `q[$]`. The `min` and `max` functions find the smallest and largest elements in an array. Note that they return a queue, not a scalar as you might expect. These methods also work for associative arrays. The `unique` method returns a queue of the unique values from the array – duplicate values are not included.

Sample 2.25 Array locator methods: `min`, `max`, `unique`

```
int f[6] = Õ{1,6,2,6,8,6};
int d[]  = Õ{2,4,6,8,10};
int q[$] = {1,3,5,7}, tq[$];

tq = q.min();                              // {1}
tq = d.max();                              // {10}
tq = f.unique();                           // {1,6,2,8}
```

You could search through an array using a `foreach`-loop, but SystemVerilog can do this in one operation with a locator method. The `with` expression tells SystemVerilog how to perform the search, as shown in Sample 2.26.

Sample 2.26 Array locator methods: `find`

```
int d[] = Õ{9,1,8,3,4,4}, tq[$];

// Find all elements greater than 3
tq = d.find with (item > 3);               // {9,8,4,4}
// Equivalent code
tq.delete();
foreach (d[i])
  if (d[i] > 3)
    tq.push_back(d[i]);

tq = d.find_index with (item > 3);         // {0,2,4,5}
tq = d.find_first with (item > 99);        // {} Ð none found
tq = d.find_first_index with (item==8);    // {2} d[2]=8
tq = d.find_last with (item==4);           // {4}
tq = d.find_last_index with (item==4);     // {5} d[5]=4
```

In a `with` clause, the name `item` is called the iterator argument and represents a single element of the array. You can specify your own name by putting it in the argument list of the array method as shown in Sample 2.27.

Sample 2.27 Declaring the iterator argument

```
tq = d.find_first with (item==4);          // These
tq = d.find_first() with (item==4);        // are
tq = d.find_first(item) with (item==4);    // all
tq = d.find_first(x) with (x==4);          // equivalent
```

Sample 2.28 shows various ways to total up a subset of the values in the array. The first total compares the item with 7. This relational returns a 1 (true) or 0 (false) and multiplies this with the array. So the sum of {9,0,8,0,0,0} is 17. The second total is computed using the ? : conditional operator.

Sample 2.28 Array locator methods

```
int count, total, d[] = õ{9,1,8,3,4,4};

count = d.sum with (item >  7);  // 2: {9, 8}
total = d.sum with ((item > 7) * item); // 17= 9+8
count = d.sum with (item <  8);  // 4: {1, 3, 4, 4}
total = d.sum with (item < 8 ? item : 0); // 12=1+3+4+4
count = d.sum with (item == 4);  // 2: {4, 4}
```

When you combine an array reduction such as `sum` using the `with` clause, the results may surprise you. In Sample 2.28, the `sum` operator totals the number of times that the expression is true. For the first statement in Sample 2.28, there are two array elements that are greater than 7 (9 and 8) and so `count` is set to 2.



The array locator methods that return an index, such as **`find_index`**, return a queue of type **`int`**, not **`integer`**. Your code may not compile if you use the wrong queue type with these statements.

2.7.3 Array Sorting and Ordering

SystemVerilog has several methods for changing the order of elements in an array. You can sort the elements, reverse their order, or shuffle the order as shown in Sample 2.29. Notice that these change the original array, unlike the array locator methods in Section 2.7.2, which create a queue to hold the results.

Sample 2.29 Sorting an array

```
int d[] =         õ{9,1,8,3,4,4};
d.reverse();  // õ{4,4,3,8,1,9}
d.sort();     // õ{1,3,4,4,8,9}
d.rsort();    // õ{9,8,4,4,3,1}
d.shuffle();  // õ{9,4,3,8,1,4}
```

The `reverse` and `shuffle` methods have no `with`-clause, and so they work on the entire array. Sample 2.30 shows how to sort a structure by sub-fields. Structures and packed structures are explained in Section 2.10.

Sample 2.30 Sorting an array of structures

```
struct packed { byte red, green, blue; } c[];
initial begin
  c = new[100];          // Allocate 100 pixels
  foreach(c[i])
    c[i] = $urandom;     // Fill with random values

  c.sort with (item.red); // sort using red only

  // sort by green value then blue
  c.sort(x) with ({x.green, x.blue});
end
```

2.7.4 Building a Scoreboard with Array Locator Methods

The array locator methods can be used to build a scoreboard. Sample 2.31 defines the `Packet` structure, then creates a scoreboard made from a queue of these structures. Section 2.9 describes how to create structures with `typedef`.

Sample 2.31 A scoreboard with array methods

```
typedef struct packed
  {bit [7:0] addr;
   bit [7:0] pr;
   bit [15:0] data; } Packet;

Packet scb[$];

function void check_addr(bit [7:0] addr);
  int intq[$];

  intq = scb.find_index() with (item.addr == addr);
  case (intq.size())
  0: $display("Addr %h not found in scoreboard", addr);
  1: scb.delete(intq[0]);
  default:
    $display("ERROR: Multiple hits for addr %h", addr);
  endcase
endfunction : check_addr
```

The `check_addr()` function in Sample 2.31 looks up an address in the scoreboard. The `find_index()` method returns an `int` queue. If the queue is empty (size==0), no match was found. If the queue has one member (size==1), a single match was found, which the `check_addr()` function deletes. If the queue has multiple members (size > 1), there are multiple packets in the scoreboard whose address matches the requested one.

A better choice for storing packet information is a class, which is described in Chap. 5. You can read more about structures in Section 2.10.

2.8 Choosing a Storage Type

Here are some guidelines for choosing the right storage type based on flexibility, memory usage, speed, and sorting. These are just rules of thumb, and results may vary between simulators.

2.8.1 Flexibility

Use a fixed-size or dynamic array if it is accessed with consecutive positive integer indices: 0, 1, 2, 3…. Choose a fixed-size array if the array size is known at compile time, or choose a dynamic array if the size is not known until run-time. For example, variable-size packets can easily be stored in a dynamic array. If you are writing routines to manipulate arrays, consider using just dynamic arrays, as one routine can work with any size dynamic array as long as the element type (`int`, `string`, etc.) matches. Likewise, you can pass a queue of any size into a routine as long as the element type matches the queue argument. Associative arrays can also be passed regardless of size. However, a routine with a fixed-size array argument only accepts arrays of the specified length.

Choose associative arrays for nonstandard indices such as widely separated values because of random values or addresses. Associative arrays can also be used to model content-addressable memories.

Queues are a good way to store values when the number of elements grows and shrinks a lot during simulation, such as a scoreboard that holds expected values.

2.8.2 Memory Usage

If you want to reduce the simulation memory usage, use 2-state elements. You should choose data sizes that are multiples of 32 bits to avoid wasted space. Simulators usually store anything smaller in a 32-bit word. For example, an array of 1,024 bytes wastes ¾ of the memory if the simulator puts each element in a 32-bit word. Packed arrays can also help conserve memory.

For arrays that hold up to a thousand elements, the type of array that you choose does not make a big difference in memory usage (unless there are many instances of these arrays). For arrays with a thousand to a million active elements, fixed-size and dynamic arrays are the most memory efficient. You may want to reconsider your algorithms if you need arrays with more than a million active elements.

Queues are slightly less efficient to access than fixed-size or dynamic arrays because of additional pointers. However, if your data set grows and shrinks often, and you

store it in a dynamic memory, you will have to manually call `new[]` to allocate memory and copy. This is an expensive operation and would wipe out any gains from using a dynamic memory.

Modeling memories larger than a few megabytes should be done with an associative array. Note that each element in an associative array can take several times more memory than a fixed-size or dynamic memory because of pointer overhead.

2.8.3 Speed

Choose your array type based on how many times it is accessed per clock cycle. For only a few reads and writes, you could use any type, as the overhead is minor compared with the DUT. As you use an array more often, its size and type matters.

Fixed-size and dynamic arrays are stored in contiguous memory, and so any element can be found in the same amount of time, regardless of array size.

Queues have almost the same access time as a fixed-size or dynamic array for reads and writes. The first and last elements can be pushed and popped with almost no overhead. Inserting or removing elements in the middle requires many elements to be shifted up or down to make room. If you need to insert new elements into a large queue, your testbench may slow down, and so consider changing how you store new elements.

When reading and writing associative arrays, the simulator must search for the element in memory. The LRM does not specify how this is done, but popular ways are hash tables and trees. These require more computation than other arrays, and therefore associative arrays are the slowest.

2.8.4 Sorting

Since SystemVerilog can sort any single-dimension array (fixed-size, dynamic, and associative arrays plus queues), you should pick based on how often the values are added to the array. If the values are received all at once, choose a fixed-size or dynamic array so that you only have to allocate the array once. If the data slowly dribbles in, choose a queue, as adding new elements to the head or tail is very efficient.

If you have unique and noncontiguous values, such as `Õ{1, 10, 11, 50}`, you can store them in an associative array by using them as an index. Using the routines `first`, `next`, and `prev`, you can search an associative array for a value and find successive values. Lists are doubly linked, and so you can find values both larger and smaller than the current value. Both of these support removing a value. However, the associative array is much faster in accessing any given element, given an index.

For example, you can use an associative array of bits to hold expected 32-bit values. When the value is created, write to that location. When you need to see if a given

value has been written, use the `exists` function. When done with an element, use `delete` to remove it from the associative array.

2.8.5 Choosing the Best Data Structure

Here are some suggestions on choosing a data structure.

- *Network packets*. Properties: fixed size, accessed sequentially. Use a fixed-size or dynamic array for fixed- or variable-size packets.
- *Scoreboard of expected values*. Properties: size not known until run-time, accessed by value, and a constantly changing size. In general, use a queue, as you are continually adding and deleting elements during simulation. If you can give every transaction a fixed id, such as 1, 2, 3, …, you could use this as an index into the queue. If your transaction is filled with random values, you can just push them into a queue and search for unique values. If the scoreboard may have hundreds of elements, and you are often inserting and deleting them from the middle, an associative array may be faster.
- *Sorted structures*. Use a queue if the data comes out in a predictable order, or an associative array if the order is unspecified. If the scoreboard never needs to be searched, just store the expected values in a mailbox, as shown in Section 7.6.
- *Modeling very large memories, greater than a million entries*. If you do not need every location, use an associative array as a sparse memory. If you do need every location, try a different approach where you do not need so much live data. Still stuck? Be sure to use 2-state values packed into 32-bits.
- *Command names or opcodes from a file*. Property: translate a string to a fixed value. Read string from a file, and then look up the commands or opcodes in an associative array using the command as a string index.

You can create an array of handles that point to objects, as shown in Chap. 5 on Basic OOP.

2.9 Creating New Types with `typedef`

You can create new types using the `typedef` statement. For example, you may have an ALU that can be configured at compile-time to use 8, 16, 24, or 32-bit operands. In Verilog you would define a macro for the operand width and another for the type as shown in Sample 2.32.

Sample 2.32 User-defined type-macro in Verilog

```
// Old Verilog style
`define OPSIZE 8
`define OPREG reg [`OPSIZE-1:0]

`OPREG op_a, op_b;
```

You are not really creating a new type; you are just performing text substitution. In SystemVerilog you create a new type with the following code. This book uses the convention that user-defined types use the suffix "`_t`."

Sample 2.33 User-defined type in SystemVerilog

```
// New SystemVerilog style
parameter OPSIZE = 8;
typedef reg [OPSIZE-1:0] opreg_t;

opreg_t op_a, op_b;
```

In general, SystemVerilog lets you copy between these basic types with no warning, either extending or truncating values if there is a width mismatch.

Note that `parameter` and `typedef` statements can be put in a package so that they can be shared across the design and testbench, as shown in Section 4.6.



One of the most useful types you can create is an unsigned, 2-state, 32-bit integer. Most values in a testbench are positive integers such as field length or number of transactions received, and so having a signed integer can cause problems. Put the definition of `uint` in a package of common definitions so that it can be used anywhere in your simulation.

Sample 2.34 Definition of `uint`

```
typedef bit [31:0] uint;     // 32-bit unsigned 2-state
typedef int unsigned uint;   // Equivalent definition
```

The syntax for defining a new array type is not obvious. You need to put the array subscripts on the new name. Sample 2.35 creates a new type, `fixed_array5`, which is a fixed array with 5 elements. It then declares an array of this type and initializes it.

Sample 2.35 User-defined array type

```
typedef int fixed_array5[5];
fixed_array5 f5;

initial begin
  foreach (f5[i])
    f5[i] = i;
end
```

2.10 Creating User-Defined Structures

One of the biggest limitations of Verilog is the lack of data structures. In SystemVerilog you can create a structure using the `struct` statement, similar to what is available in C. However, a `struct` has just a subset of the functionality of a class, and so use a class instead for your testbenches, as shown in Chap. 5. Just as a Verilog module combines both data (signals) and code (always/initial blocks plus routines), a class combines data and routines to make an entity that can be easily debugged and reused. A `struct` just groups data fields together. Without the code that manipulates the data, you are only creating half of the solution.

Since a `struct` is just a collection of data, it can be synthesized. If you want to model a complex data type, such as a pixel, in your design code, put it in a `struct`. This can also be passed through module ports. Eventually, when you want to generate constrained random data, look to classes.

2.10.1 Creating a `struct` and a New Type

You can combine several variables into a structure. Sample 2.36 creates a structure called `pixel` that has three unsigned bytes for red, green, and blue.

Sample 2.36 Creating a single pixel type

```
struct {bit [7:0] r, g, b;} pixel;
```

The problem with the preceding declaration is that it creates a single pixel of this type. To be able to share pixels using ports and routines, you should create a new type instead, as shown in Sample 2.37.

Sample 2.37 The pixel `struct`

```
typedef struct {bit [7:0] r, g, b;} pixel_s;
pixel_s my_pixel;
```

Use the suffix "`_s`" when declaring a `struct`. This makes it easier to spot user-defined types, simplifying the process of sharing and reusing code.

2.10.2 Initializing a Structure

You can assign multiple values to a struct just like an array, either in the declaration or in a procedural assignment. Just surround the values with an apostrophe and braces, as shown in Sample 2.38.

Sample 2.38 Initializing a struct

```
initial begin
  typedef struct {int a;
                  byte b;
                  shortint c;
                  int d;} my_struct_s;
  my_struct_s st = '{32'haaaa_aaaad,
                     8'hbb,
                     16'hcccc,
                     32'hdddd_dddd};

  $display("str = %x %x %x %x ", st.a, st.b, st.c, st.d);
end
```

2.10.3 Making a Union of Several Types

In hardware, the interpretation of a set of bits in a register may depend on the value of other bits. For example, a processor instruction may have many layouts based on the opcode. Immediate-mode operands might store a literal value in the operand field. This value may be decoded differently for integer instructions than for floating point instructions. Sample 2.39 stores both the integer `i` and the real `f` in the same location.

Sample 2.39 Using `typedef` to create a union

```
typedef union { int i; real f; } num_u;
num_u un;
un.f = 0.0; // set value in floating point format
```

Use the suffix "_u" when declaring a union.



Unions are useful when you frequently need to read and write a register in several different formats. However, don't go overboard, especially just to save memory. Unions may help squeeze a few bytes out of a structure, but at the expense of having to create and maintain a more complicated data structure. Instead, make a flat class with a discriminant variable, as shown in Section 8.4.4. This "kind" variable indicates which type of transaction you have, and thus which fields to read, write, and randomize. If you just need an array of values, plus all the bits, use a packed array as described in Section 2.2.6

2.10.4 Packed Structures

SystemVerilog allows you more control in how bits are laid out in memory by using packed structures. A packed structure is stored as a contiguous set of bits with no unused space. The `struct` for a pixel in Sample 2.37 used three values, and so it is stored in three longwords, even though it only needs three bytes. You can specify that it should be packed into the smallest possible space.

Sample 2.40 Packed structure

```
typedef struct packed {bit [7:0] r, g, b;} pixel_p_s;
pixel_p_s my_pixel;
```

Packed structures are used when the underlying bits represent a numerical value, or when you are trying to reduce memory usage. For example, you could pack together several bit-fields to make a single register. Or you might pack together the opcode and operand fields to make a value that contains an entire processor instruction.

2.10.5 Choosing Between Packed and Unpacked Structures

When you are trying to choose between packed and unpacked structures, consider how the structure is most commonly used, and the alignment of the elements. If you plan on making aggregate operations on the structure, such as copying the entire structure, a packed structure is more efficient. However, if your code accesses the individual members more than the entire structure, use an unpacked structure. The difference in performance is greater if the elements are not aligned on byte boundaries, have sizes that don't match the typical byte, or have word instructions used by processors. Reading and writing elements with odd sizes in a packed structure requires expensive shift and mask operations.

2.11 Type Conversion

The proliferation of data types in SystemVerilog means that you will need to convert between them. If the layout of the bits between the source and destination variables are the same, such as an integer and enumerated type, cast between the two values. If the bit layouts differ, such as an array of bytes and words, use the streaming operators to rearrange the bits.

2.11.1 The Static Cast

The static cast operation converts between two types with no checking of values. You specify the destination type, an apostrophe, and the expression to be converted as shown in Sample 2.41. Note that Verilog has always implicitly converted between types such as integer and real, and also between different width vectors.

Sample 2.41 Converting between int and real with static cast

```
int i;
real r;

i = int '(10.0 - 0.1); // cast is optional
r = real'(42);         // cast is optional
```

2.11.2 The Dynamic Cast

The dynamic cast, `$cast`, allows you to check for out-of-bounds values. See Section 2.12.3 for an explanation and example with enumerated types.

2.11.3 Streaming Operators

When used on the right side of an assignment, the streaming operators << and >> take an expression, structure, or array, and packs it into a stream of bits. The >> operator streams data from left to right while << streams from right to left, as shown in Sample 2.42. You can also give a slice size, which is used to break up the source before being streamed. You can not assign the bit stream result directly to an unpacked array. Instead, use the streaming operators on the left side of an assignment to unpack the bit stream into an unpacked array.

Sample 2.42 Basic streaming operator

```
initial begin
  int h;
  bit [7:0] b, g[4], j[4] = '{8'ha, 8'hb, 8'hc, 8'hd};
  bit [7:0] q, r, s, t;

  h = { >> {j}};          // 0a0b0c0d - pack array into int
  h = { << {j}};          // b030d050 reverse bits
  h = { << byte {j}};     // 0d0c0b0a reverse bytes
  g = { << byte {j}};     // 0d, 0c, 0b, 0a unpack into array
  b = { << {8'b0011_0101}};   // 1010_1100 reverse bits
  b = { << 4 {8'b0011_0101}}; // 0101_0011 reverse nibble
  {>> {q, r, s, t}} = j; // Scatter j into bytes
  h = {>>{t, s, r, q}};  // Gather bytes into h
end
```

You could do the same operations with many concatenation operators, {}, but the streaming operators are more compact and easier to read.

If you need to pack or unpack arrays, use the streaming operator to convert between arrays of different element sizes. For instance, you can convert an array of bytes to an array of words. You can use fixed size arrays, dynamic arrays, and queues. Sample 2.43 converts between queues, but would also work with dynamic arrays. Array elements are automatically allocated as needed.

Sample 2.43 Converting between queues with streaming operator

```
initial begin
  bit [15:0] wq[$] = {16'h1234, 16'h5678};
  bit [7:0]  bq[$];

  // Convert word array to byte
  bq = { >> {wq}};  // 12 34 56 78

  // Convert byte array to words
  bq = {8'h98, 8'h76, 8'h54, 8'h32};
  wq = { >> {bq}};  // 9876 5432
end
```



A common mistake when streaming between arrays is mismatched array subscripts. The word subscript `[256]` in an array declaration is equivalent to `[0:255]`, not `[255:0]`. Since many arrays are declared with the word subscripts `[high:low]`, streaming them to an array with the subscript `[size]` would result in the elements ending up in reverse order. Likewise, streaming an unpacked array declared as `bit [7:0] src[255:0]` to the packed array declared as `bit [7:0] [255:0] dst` will scramble the order of values. The correct declaration for a packed array of bytes is `bit [255:0] [7:0] dst`.

You can also use the streaming operator to pack and unpack structures, such as an ATM cell, into an array of bytes. In Sample 2.44, a structure is streamed into a dynamic array of bytes, and then the byte array is streamed back into the structure.

Sample 2.44 Converting between a structure and array with streaming operators

```
initial begin
  typedef struct {int a;
                  byte b;
                  shortint c;
                  int d;} my_struct_s;
  my_struct_s st = '{32'haaaa_aaaa,
                     8'hbb,
                     16'hcccc,
                     32'hdddd_dddd};
  byte b[];

  // Covert from struct to byte array
  b = { >> {st}};      // {aa aa aa aa bb cc cc dd dd dd dd}

  // Convert from byte array to a struct
  b = '{8'h11, 8'h22, 8'h33, 8'h44, 8'h55, 8'h66, 8'h77,
        8'h88, 8'h99, 8'haa, 8'hbb};
  st = { >> {b}};      // st = 11223344, 55, 6677, 8899aabb
end
```

2.12 Enumerated Types

Before enumerated types, you have to use text macros. Their global scope is too broad, and in most cases are visible in the debugger. An enumeration creates a strong variable type that is limited to a set of specified names, such as the instruction opcodes or state machine values. For example, the names ADD, MOVE, or ROTW make your code easier to write and maintain than if you had used literals such as `8Õh01` or macros. Another alternative for defining constants is a parameter. These are fine for individual values, but an enumerated type automatically gives a unique value to every name in the list.

The simplest enumerated type declaration contains a list of constant names and one or more variables as shown in Sample 2.45. This creates an anonymous enumerated type, but it cannot be used for any other variables than the ones in this declaration.

Sample 2.45 A simple enumerated type

```
enum {RED, BLUE, GREEN} color;
```

In general you want to create a named enumerated type to easily declare multiple variables, especially if these are used as routine arguments or module ports. You first create the enumerated type, and then the variables of this type. You can get the string representation of an enumerated variable with the built-in function `name`(), as shown in Sample 2.46.

Sample 2.46 Enumerated types

```
// Create data type for values 0, 1, 2
typedef enum {INIT, DECODE, IDLE} fsmstate_e;
fsmstate_e pstate, nstate;   // declare typed variables

initial begin
  case (pstate)
    IDLE:    nstate = INIT;    // data assignment
    INIT:    nstate = DECODE;
    default: nstate = IDLE;
  endcase
  $display("Next state is %s",
           nstate.name());     // Display symbolic state name
end
```

Use the suffix "_e" when declaring an enumerated type.

2.12.1 Defining Enumerated Values

The actual values default to integers starting at 0 and then increase. You can choose your own enumerated values. The code in Sample 2.47 uses the default value of 0 for `INIT`, then 2 for `DECODE`, and 3 for `IDLE`.

Sample 2.47 Specifying enumerated values

```
typedef enum {INIT, DECODE=2, IDLE} fsmtype_e;
```

Enumerated constants, such as `INIT` in Sample 2.47, follow the same scoping rules as variables. Consequently, if you use the same name in several enumerated types (such as `INIT` in different state machines), they have to be declared in different scopes such as modules, program blocks, packages, routines, or classes.



An enumerated type is stored as `int` unless you specify otherwise. Be careful when assigning values to enumerated constants, as the default value of an `int` is 0. In Sample 2.48, `position` is initialized to 0, which is not a legal `ordinal_e` variable. This behavior is not a tool bug – it is how the language is specified. So always specify an enumerated constant with the value of 0, as shown in Sample 2.49, just to catch the testbench error.

Sample 2.48 Incorrectly specifying enumerated values

```
typedef enum {FIRST=1, SECOND, THIRD} ordinal_e;
ordinal_e position;
```

Sample 2.49 Correctly specifying enumerated values

```
typedef enum {BAD_O=0, FIRST=1, SECOND, THIRD} ordinal_e;
ordinal_e position;
```

2.12.2 Routines for Enumerated Types

SystemVerilog provides several functions for stepping through enumerated types.

- `first()` returns the first member of the enumeration.
- `last()` returns the last member of the enumeration.
- `next()` returns the next element of the enumeration.
- `next(N)` returns the *N*th next element.
- `prev()` returns the previous element of the enumeration.
- `prev(N)` returns the *N*th previous element.

The functions `next` and `prev` wrap around when they reach the beginning or end of the enumeration.

Note that there is no easy way to write a `for`-loop that steps through all members of an enumerated type if you use an enumerated loop variable. You get the starting member with `first` and the next member with `next`. The problem is creating a comparison for the final iteration through the loop. If you use the test `current!=current.last`, the loop ends before using the last value. If you use `current<=current.last`, you get an infinite loop, as `next` never gives you a value that is greater than the final value. This is similar to trying to make a `for`-loop that steps through the values 0.3 with index declared as `bit [1:0]`. The loop will never exit!

You can use a `do...while` loop to step through all the values, as shown in Sample 2.50.

Sample 2.50 Stepping through all enumerated members

```
typedef enum {RED, BLUE, GREEN} color_e;
color_e color;
color = color.first;
do
  begin
  $display("Color = %0d/%s", color, color.name);
  color = color.next;
  end
while (color != color.first);  // Done at wrap-around
```

2.12.3 Converting to and from Enumerated Types

The default type for an enumerated type is `int` (2-state). You can take the value of an enumerated variable and assign it to a nonenumerated variable such as an `int` with a simple assignment. SystemVerilog does not, however, let you store an integer value in an `enum` without explicitly changing the type. Instead, it requires you to explicitly cast the value to make you realize that you could be writing an out-of-bounds value.

Sample 2.51 Assignments between integers and enumerated types

```
typedef enum {RED, BLUE, GREEN} COLOR_E;
COLOR_E color, c2;
int c;

initial begin
  color = BLUE;          // Set to known good value
  c = color;             // Convert from enum to int (1)
  c++;                   // Increment int (2)
  if (!$cast(color, c)) // Cast int back to enum
    $display("Cast failed for c=%0d", c);
  $display("Color is %0d / %s", color, color.name);
  c++;                   // 3 is out-of-bounds for enum
  c2 = COLOR_EÕ(c);      // No type checking
  $display("Color is %0d / %s", color, color.name);
  end
```

When called as a function as shown in Sample 2.51, `$cast`() tried to assign the right value to the left variable. If the assignment succeeds, `$cast()` returns 1. If the assignment fails because of an out-of-bounds value, no assignment is made and the function returns 0. If you use `$cast`() as a task and the operation fails, SystemVerilog prints an error.

You can also cast the value using the `typeÕ(val)` as shown in the example, but this does not do any type checking, and so the result may be out-of-bounds. For example,

after the static cast in Sample 2.51, `c2` has an out-of-bounds value. You should avoid this style.

2.13 Constants

There are several types of constants in SystemVerilog. The classic Verilog way to create a constant is with a text macro. On the plus side, macros have global scope and can be used for bit field definitions and type definitions. On the negative side, macros are global, so that they can cause conflicts if you just need a local constant. Lastly, a macro requires the ` character so that it is recognized and expanded by the compiler.

In SystemVerilog, parameters can be declared in a package and so they can be used across multiple modules. This approach can replace many Verilog macros that were just being used as constants. You can use a `typedef` to replace those clunky macros. The next choice is a `parameter`. A Verilog `parameter` was loosely typed and was limited in scope to a single module. Verilog-2001 added typed parameters, but the limited scope kept parameters from being widely used.

SystemVerilog also supports the `const` modifier that allows you to make a variable that can be initialized in the declaration but not written by procedural code.

Sample 2.52 Declaring a `const` variable

```
initial begin
  const byte colon = ":";
  ...
end
```

In Sample 2.52, the value of `colon` is initialized when the `initial` block is entered. In the next chapter, Sample 3.10 shows a `const` routine argument.

2.14 Strings

If you have ever tried to use a Verilog `reg` variable to hold a string of characters, your suffering is over. The SystemVerilog `string` type holds variable-length strings. An individual character is of type `byte`. The elements of a string of length *N* are numbered 0 to *N*-1. Note that, unlike C, there is no null character at the end of a string, and any attempt to use the character "\0" is ignored. Memory for strings is dynamically allocated, so you do not have to worry about running out of space to store the string.

Sample 2.53 shows various string operations. The function `getc(`*N*`)` returns the byte at location *N*, while `toupper` returns an upper-case copy of the string and `tolower` returns a lowercase copy. The curly braces {} are used for concatenation. The task `putc(`*M*`, `*C*`)` writes a byte *C* into a string at location *M*, which must be between 0 and

the length as given by `len`. The `substr(start,end)` function extracts characters from location `start` to `end`.

Sample 2.53 String methods

```
string s;

initial begin
  s = "IEEE ";
  $display(s.getc(0));       // Display: 73 (ÔIÕ)
  $display(s.tolower());     // Display: ieee

  s.putc(s.len()-1, "-");    // change Õ Õ-> Õ-Õ
  s = {s, "P1800"};          // "IEEE-P1800"

  $display(s.substr(2, 5)); // Display: EE-P

  // Create temporary string, note format
  my_log_rtn($psprintf("%s %5d", s, 42));
end

task my_log(string message);
  // Print a message to a log
  $display("@%0t: %s", $time, message);
endtask
```

Note how useful dynamic strings can be. In other languages such as C, you have to keep making temporary strings to hold the result from a function. In Sample 2.53, the `$psprintf()` function is used instead of `$sformat()`, from Verilog-2001. This new function returns a formatted temporary string that, as shown above, can be passed directly to another routine. This saves you from having to declare a temporary string and passing it between the formatting statement and the routine call.

2.15 Expression Width

A prime source for unexpected behavior in Verilog has been the width of expressions. Sample 2.54 adds 1+1 using four different styles. Addition `A` uses two 1-bit variables, and so with this precision 1+1=0. Addition `B` uses 8-bit precision because there is an 8-bit variable on the left side of the assignment. In this case, 1+1=2. Addition `C` uses a dummy constant to force SystemVerilog to use 2-bit precision. Lastly, in addition `D`, the first value is cast to be a 2-bit value with the cast operator, and so 1+1=2.

Sample 2.54 Expression width depends on context

```
bit [7:0] b8;
bit one = 1Õb1;                 // Single bit
$displayb(one + one);           // A: 1+1 = 0

b8 = one + one;                 // B: 1+1 = 2
$displayb(b8);

$displayb(one + one + 2Õb0);  // C: 1+1 = 2 with constant

$displayb(2Õ(one) + one);     // D: 1+1 = 2 with cast
```

There are several tricks you can use to avoid this problem. First, avoid situations where the overflow is lost, as in addition `A`. Use a temporary, such as `b8`, with the desired width. Or, you can add another value to force the minimum precision, such as `2Õb0`. Lastly, in SystemVerilog, you can cast one of the variables to the desired precision.

2.16 Conclusion

SystemVerilog provides many new data types and structures so that you can create high-level testbenches without having to worry about the bit-level representation. Queues work well for creating scoreboards for which you constantly need to add and remove data. Dynamic arrays allow you to choose the array size at run-time for maximum testbench flexibility. Associative arrays are used for sparse memories and some scoreboards with a single index. Enumerated types make your code easier to read and write by creating groups of named constants.

Don't go off and create a procedural testbench with just these constructs. Explore the OOP capabilities of SystemVerilog in Chap. 5 to learn how to design code at an even higher level of abstraction, thus creating robust and reusable code.

Chapter 3

Procedural Statements and Routines

As you verify your design, you need to write a great deal of code, most of which is in tasks and functions. SystemVerilog introduces many incremental improvements to make this easier by making the language look more like C, especially around argument passing. If you have a background in software engineering, these additions should be very familiar.

3.1 Procedural Statements

SystemVerilog adopts many operators and statements from C and C++. You can declare a loop variable inside a `for`-loop that then restricts the scope of the loop variable and can prevent some coding bugs. The auto-increment `++` and auto-decrement `--` operators are available in both pre- and post-forms. If you have a label on a `begin` or `fork` statement, you can put the same label on the matching `end` or `join` statement. This makes it easier to match the start and finish of a block. You can also put a label on other SystemVerilog end statements such as `endmodule`, `endtask`, `endfunction`, and others that you will learn in this book. Sample 3.1 demonstrates some of the new constructs.

Sample 3.1 New procedural statements and operators

```
initial
  begin : example
  integer array[10], sum, j;

  // Declare i in for statement
  for (int i=0; i<10; i++)       // Increment i
    array[i] = i;

  // Add up values in the array
  sum = array[9];
  j=8;
  do                             // do...while loop
    sum += array[j];             // Accumulate
  while (j--);                   // Test if j=0
  $display("Sum=%4d", sum);      // %4d - specify width
end : example                    // End label
```

Two new statements help with loops. First, if you are in a loop, but want to skip over the rest of the statements and do the next iteration, use `continue`. If you want to leave the loop immediately, use `break`.

The loop in Sample 3.2 reads commands from a file using the file I/O system tasks that are part of Verilog-2001. If the command is just a blank line, the code does a `continue`, skipping any further processing of the command. If the command is "done," the code does a `break` to terminate the loop.

Sample 3.2 Using `break` and `continue` while reading a file

```
initial begin
  bit [127:0] cmd;
  int file, c;

  file = $fopen("commands.txt", "r");
  while (!$feof(file)) begin
    c = $fscanf(file, "%s", cmd);
    case (cmd)
      "":     continue;    // Blank line - skip to loop end
      "done": break;       // Done - leave loop
      ...                  // Process other commands here
    endcase // case(cmd)
    end
  $fclose(file);
end
```

3.2 Tasks, Functions, and Void Functions

Verilog makes a very clear differentiation between tasks and functions. The most important difference is that a task can consume time, whereas a function cannot. A function cannot have a delay, `#100`, a blocking statement such as `@(posedge clock)` or `wait(ready)`, or call a task. Additionally, a Verilog function must return a value and the value must be used, as in an assignment statement.

SystemVerilog relaxes this rule a little in that a function can call a task, but only in a thread spawned with the `fork...join_none` statement, which is described in Section 7.1.



If you have a SystemVerilog task that does not consume time, you should make it a `void function`, which is a function that does not return a value. Now it can be called from any task or function. For maximum flexibility, any debug routine should be a void function rather than a task so that it can be called from any task or function. Sample 3.3 prints values from a state machine.

Sample 3.3 Void function for debug

```
function void print_state(...);
  $display("@%0t: state = %s", $time, cur_state.name());
endfunction
```

In SystemVerilog, if you want to call a function and ignore its return value, cast the result to `void`, as shown in Sample 3.4. Some simulators, such as VCS, allow you to ignore the return value without using the above `void` syntax.

Sample 3.4 Ignoring a function's return value

```
voidÕ($fscanf(file, "%d", i));
```

3.3 Task and Function Overview

SystemVerilog makes several small improvements to tasks and functions to make them look more like C or C++ routines. In general, a routine definition or call with no arguments does not need the empty parentheses (). This book includes them for added clarity.

3.3.1 Routine `begin...end` Removed

The first improvement you may notice in SystemVerilog routines is that `begin...end` blocks are optional, while Verilog-1995 required them on all but single-line routines. The `task` / `endtask` and `function` / `endfunction` keywords are enough to define the routine boundaries, as show in Sample 3.5.

Sample 3.5 Simple task without `begin...end`

```
task multiple_lines;
  $display("First line");
  $display("Second line");
endtask : multiple_lines
```

3.4 Routine Arguments

Many of the SystemVerilog improvements for routines make it easier to declare arguments and expand the ways you can pass values to and from a routine.

3.4.1 C-Style Routine Arguments

SystemVerilog and Verilog-2001 allow you to declare task and function arguments more cleanly and with less repetition. The following Verilog task requires you to declare some arguments twice: once for the direction, and once for the type, as shown in Sample 3.6.

Sample 3.6 Verilog-1995 routine arguments

```
task mytask2;
  output [31:0] x;
  reg    [31:0] x;
  input         y;
  ...
endtask
```

With SystemVerilog, you can use the less verbose C-style, shown in Sample 3.7. Note that you should use the universal input type of `logic`.

Sample 3.7 C-style routine arguments

```
task mytask1 (output logic [31:0] x,
              input  logic y);
...
endtask
```

3.4.2 Argument Direction

You can take even more shortcuts with declaring routine arguments. The direction and type default to "input logic" and are sticky, and so you don't have to repeat these for similar arguments. Sample 3.8 shows a routine header written using the Verilog-1995 style and SystemVerilog data types.

Sample 3.8 Verbose Verilog-style routine arguments

```
task T3;
  input a, b;
  logic a, b;
  output [15:0] u, v;
  bit [15:0] u, v;
  ...
endtask
```

You could rewrite this as shown in Sample 3.9.

Sample 3.9 Routine arguments with sticky types

```
task T3(a, b, output bit [15:0] u, v);
```

The arguments `a` and `b` are input logic, 1-bit wide. The arguments `u` and `v` are 16-bit output bit types. Now that you know this, don't depend on the defaults, as your code will be infested with subtle and hard to find bugs, as explained in Section 3.4.6. Always declare the type and direction for every routine argument.

3.4.3 Advanced Argument Types

Verilog had a simple way to handle arguments: an `input` or `inout` was copied to a local variable at the start of the routine, whereas an `output` or `inout` was copied when the routine exited. No memories could be passed into a Verilog routine, except scalarscan.

In SystemVerilog, you can specify that an argument is passed by reference, rather than copying its value. This argument type, `ref`, has several benefits over `input`, `output`, and `inout`. First, you can now pass an array into a routine.

Sample 3.10 Passing arrays using `ref` and `const`

```
function void print_checksum (const ref bit [31:0] a[]);
  bit [31:0] checksum = 0;
  for (int i=0; i<a.size(); i++)
    checksum ^= a[i];
  $display("The array checksum is %0d", checksum);
endfunction
```

SystemVerilog allows you to pass array arguments without the `ref` direction, but the array is copied onto the stack, an expensive operation for all but the smallest arrays.

The SystemVerilog LRM states that `ref` arguments can only be used in routines with automatic storage. If you specify the `automatic` attribute for programs and module, all the routines inside are automatic. See Section 3.6 for more details on storage.

Sample 3.10 also shows the `const` modifier. As a result, the array `a` points to the array in the routine call, but the contents of the array cannot be modified. If you try to change the contents, the compiler prints an error.



Always use `ref` when passing arrays to a routine for best performance. If you don't want the routine to change the array values, use the `const ref` type, which causes the compiler to check that your routine does not modify the array.

The second benefit of `ref` arguments is that a task can modify a variable and is instantly seen by the calling function. This is useful when you have several threads executing concurrently and want a simple way to pass information. See Chap. 7 for more details on using `fork-join`.

In Sample 3.11, the `thread2` block in the initial block can access the data from memory as soon as `bus.enable` is asserted, even though the `bus_read` task does not return until the bus transaction completes, which could be several cycles later. The `data` argument is passed as `ref`, and as a result, the `@data` statement triggers as soon as `data` changes in the task. If you had declared `data` as `output`, the `@data` statement would not trigger until the end of the bus transaction.

Sample 3.11 Using `ref` across threads

```
task bus_read(input logic [31:0] addr,
              ref   logic [31:0] data);

  // Request bus and drive address
  bus.request = 1'b1;
  @(posedge bus.grant) bus.addr = addr;

  // Wait for data from memory
  @(posedge bus.enable) data = bus.data;

  // Release bus and wait for grant
  bus.request = 1'b0;
  @(negedge bus.grant);
endtask

logic [31:0] addr, data;

initial
  fork
    bus_read(addr, data);
    thread2: begin
      @data;  // Trigger on data change
      $display("Read %h from bus", data);
    end
  join
```

3.4.4 Default Value for an Argument

As your testbench grows in sophistication, you may want to add additional controls to your code but not break existing code. For the function in Sample 3.10, you might want to print a checksum of just the middle values of the array. However, you don't want to go back and rewrite every call to add extra arguments. In SystemVerilog you can specify a default value that is used if you leave out an argument in the call. Sample 3.12 adds `low` and `high` arguments to the `print_checksum` function so that you can print a checksum of a range of values.

Sample 3.12 Function with default argument values

```
function void print_checksum(ref bit [31:0] a[],
                             input bit [31:0] low = 0,
                             input int high = -1);
  bit [31:0] checksum = 0;

  if (high == -1 || high >= a.size())
    high = a.size()-1;

  for (int i=low; i<=high; i++)
    checksum += a[i];
  $display("The array checksum is %0d", sum);
endfunction
```

You can call this function in the following ways, as shown in Sample 3.13. Note that the first call is compatible with both versions of the `print_checksum` routine.

Sample 3.13 Using default argument values

```
print_checksum(a);       // Checksum a[0:size()-1] Ð default
print_checksum(a, 2, 4); // Checksum a[2:4]
print_checksum(a, 1);    // Start at 1
print_checksum(a,, 2);   // Checksum a[0:2]
print_checksum();        // Compile error: a has no default
```

Using a default value of –1 (or any out-of-range value) is a good way to see if the call specified a value.

A Verilog `for`-loop always executes the initialization (`int i=low`), and test (`i<=high`) before starting the loop. Thus, if you accidently passed a `low` value that was larger than `high` or the array size, the `for`-loop would never execute the body.

3.4.5 Passing Arguments by Name

You may have noticed in the SystemVerilog LRM that the arguments to a task or function are sometimes called “ports,” just like the connections for a module. If you have a task or function with many arguments, some with default values, and you only want to set a few of those arguments, you can specify a subset by specifying the name of the routine argument with a port-like syntax, as shown in Sample 3.14.

Sample 3.14 Binding arguments by name

```
task many (input int a=1, b=2, c=3, d=4);
  $display("%0d %0d %0d %0d", a, b, c, d);
endtask

initial begin          // a  b  c  d
  many(6, 7, 8, 9);    // 6  7  8  9  Specify all values
  many();              // 1  2  3  4  Use defaults
  many(.c(5));         // 1  2  5  4  Only specify c
  many(, 6, .d(8));    // 1  6  3  8  Mix styles
end
```

3.4.6 Common Coding Errors



The most common coding mistake that you are likely to make with a routine is forgetting that the argument type is sticky with respect to the previous argument, and that the default type for the first argument is a single-bit input. Start with the simple task header in Sample 3.15.

Sample 3.15 Original task header

```
task sticky(int a, b);
```

The two arguments are input integers. As you are writing the task, you realize that you need access to an array, and so you add a new array argument, and use the `ref` type so that it does not have to be copied. Your routine header now looks like Sample 3.16.

Sample 3.16 Task header with additional array argument

```
task sticky(ref int array[50],
            int a, b);      // What direction are these?
```

What argument types are `a` and `b`? They take the direction of the previous argument `ref`. Using `ref` for a simple variable such as an `int` is not usually needed, but you would not get even a warning from the compiler, and thus would not realize that you were using the wrong direction.

If any argument to your routine is something other than the default input type, specify the direction for all arguments as shown in Sample 3.17.

Sample 3.17 Task header with additional array argument

```
task sticky(ref   int array[50],
            input int a, b);  // Be explicit
```

3.5 Returning from a Routine

Verilog had a primitive way to end a routine; after you executed the last statement in a routine, it returned to the calling code. In addition, a function returned a value by assigning that value to a variable with the same name as the function.

3.5.1 The Return Statement

SystemVerilog adds the `return` statement to make it easier for you to control the flow in your routines. The task in Sample 3.18 needs to return early because of error checking. Otherwise, it would have to put the rest of the task in an `else` clause, which would cause more indentation and be more difficult to read.

Sample 3.18 Return in a task

```
task load_array(int len, ref int array[]);
  if (len <= 0) begin
    $display("Bad len");
    return;
  end

  // Code for the rest of the task
  ...
endtask
```

The `return` statement in Sample 3.19 can simplify your functions.

Sample 3.19 Return in a function

```
function bit transmit(...);
  // Send transaction
  ...
  return ~ifc.cb.error; // Return status: 0=error
endfunction
```

3.5.2 Returning an Array from a Function

Verilog routines could only return a simple value such as a bit, integer, or vector. If you wanted to compute and return an array, there was no simple way. In SystemVerilog, a function can return an array, using several techniques.

The first way is to define a type for the array, and then use that in the function declaration. Sample 3.20 uses the array type from Sample 2.35, and creates a function to initialize the array.

Sample 3.20 Returning an array from a function with a typedef

```
typedef int fixed_array5[5];
fixed_array5 f5;

function fixed_array5 init(int start);
  foreach (init[i])
    init[i] = i + start;
endfunction

initial begin
  f5 = init(5);
  foreach (f5[i])
    $display("f5[%0d] = %0d", i, f5[i]);
end
```

One problem with the preceding code is that the function `init` creates an array, which is copied into the array `f5`. If the array was large, this could be a large performance problem.

The alternative is to pass the routine by reference. The easiest way is to pass the array into the function as a `ref` argument, as shown in Sample 3.21.

Sample 3.21 Passing an array to a function as a ref argument

```
function void init(ref int f[5], input int start);
  foreach (f[i])
    f[i] = i + start;
endfunction

int fa[5];
initial begin
  init(fa, 5);
  foreach (fa[i])
    $display("fa[%0d] = %0d", i, fa[i]);
end
```

The last way for a function to return an array is to wrap the array inside of a class, and return a handle to an object. Chap. 5 describes classes, objects, and handles.

3.6 Local Data Storage

When Verilog was created in the 1980s, it was tightly tied to describing hardware. As a result, all objects in the language were statically allocated. In particular, routine arguments and local variables were stored in a fixed location, rather than pushing them on a stack like other programming languages. Why try to model dynamic code such as a recursive routine when there is no way to build this in silicon? However,

software engineers verifying the designs, who were used to the behavior of stack-based languages such as C, were bitten by these subtle bugs, and were thus limited in their ability to create complex testbenches with libraries of routines.

3.6.1 Automatic Storage

In Verilog-1995, if you tried to call a task from multiple places in your testbench, the local variables shared common, static storage, and so the different threads stepped on each other's values. In Verilog-2001 you can specify that tasks, functions, and modules use automatic storage, which causes the simulator to use the stack for local variables.



In SystemVerilog, routines still use static storage by default, for both modules and program blocks. You should always make program blocks (and their routines) use automatic storage by putting the `automatic` keyword in the program statement. In Chap. 4 you will learn about `program` blocks that hold the testbench code. Section 7.1.6 shows how automatic storage helps when you are creating multiple threads.

Sample 3.22 shows a task to monitor when data are written into memory.

Sample 3.22 Specifying automatic storage in program blocks

```
program automatic test;
  task wait_for_mem(input [31:0] addr, expect_data,
                    output success);
    while (bus.addr !== addr)
      @(bus.addr);
    success = (bus.data == expect_data);
  endtask
...
endprogram
```

You can call this task multiple times concurrently, as the `addr` and `expect_data` arguments are stored separately for each call. Without the `automatic` modifier, if you called `wait_for_mem` a second time while the first was still waiting, the second call would overwrite the two arguments.

3.6.2 Variable Initialization



A similar problem occurs when you try to initialize a local variable in a declaration, as it is actually initialized before the start of simulation. The general solution is to avoid initializing a variable in a declaration to anything other than a constant. Use a separate assignment statement to give you better control over when initialization is done.

The task in Sample 3.23 looks at the bus after five cycles and then creates a local variable and attempts to initialize it to the current value of the address bus.

Sample 3.23 Static initialization bug

```
program initialization; // Buggy version
  task check_bus;
    repeat (5) @(posedge clock);
    if (bus_cmd == ÔREAD) begin
      // When is local_addr initialized?
      logic [7:0] local_addr = addr<<2;  // Bug
      $display("Local Addr = %h", local_addr);
    end
  endtask
endprogram
```

The bug is that the variable `local_addr` is statically allocated, and so it is actually initialized at the start of simulation, not when the `begin...end` block is entered. Once again, the solution is to declare the program as `automatic` as shown in Sample 3.24.

Sample 3.24 Static initialization fix: use `automatic`

```
program automatic initialization; // Bug solved
...
endprogram
```

Additionally, you can avoid this by never initializing a variable in the declaration, but this is harder to remember, especially for C programmers. Sample 3.25 show the recommended style of separating the declaration and initialization.

Sample 3.25 Static initialization fix: break apart declaration and initialization

```
logic [7:0] local_addr
local_addr = addr << 2;  // Bug
```

3.7 Time Values

SystemVerilog has several new constructs to allow you to unambiguously specify time values in your system.

3.7.1 Time Units and Precision

When you rely on the `Ôtimescale` compiler directive, you must compile the files in the proper order to be sure all the delays use the proper scale and precision. The `timeunit` and `timeprecision` declarations eliminate this ambiguity by precisely specifying the values for every module. Sample 3.26 shows these declarations. Note

that if you use these instead of Ôtimescale, you must put them in every module that has a delay.

3.7.2 Time Literals

SystemVerilog allows you to unambiguously specify a time value plus units. Your code can use delays such as `0.1ns` or `20ps`. Just remember to use `timeunit` and `timeprecision` or Ôtimescale. You can make your code even more time aware by using the classic Verilog `$timeformat()`, `$time`, and `$realtime` system tasks. The four arguments to $timef ormat are the scaling factor (–9 for nanoseconds, –12 for picoseconds), the number of digits to the right of the decimal point, a string to print after the time value, and the minimum field width.

Sample 3.26 shows various delays and the result from printing the time when it is formatting by `$timeformat()` and the `%t` specifier.

Sample 3.26 Time literals and `$timeformat`

```
module timing;
  timeunit 1ns;
  timeprecision 1ps;
  initial begin
    $timeformat(-9, 3, "ns", 8);
    #1     $display("%t", $realtime); // 1.000ns
    #2ns   $display("%t", $realtime); // 3.000ns
    #0.1ns $display("%t", $realtime); // 3.100ns
    #41ps  $display("%tÓ, $realtime); // 3.141ns
  end
endmodule
```

3.7.3 Time and Variables

You can store time values in variables and use them in calculations and delays. The values are scaled and rounded according to the current time scale and precision. Variables of type `time` cannot hold fractional delays as they are just 64-bit integers, and so delays will be rounded. You should use `real` variables if this is a problem.

Sample 3.27 shows how real variables are able to retain accurate values and are only rounded when used as a delay.

Sample 3.27 Time variables and rounding

```
Ôtimescale 1ps/1ps
module ps;
  initial begin
    real rdelay = 800fs;    // Stored as 0.800
    time tdelay = 800fs;    // Rounded to 1
    $timeformat(-15, 0, "fs", 5);
    #rdelay;                // Delay rounded to 1ps
    $display("%t", rdelay); // "800fs"
    #tdelay;                // Delay another 1ps
    $display("%t", tdelay); // "1000fs"
   end

endmodule
```

3.7.4 $time vs. $realtime

The system task `$time` returns an integer scaled to the time precision of the current module, but missing any fractional units, while `$realtime` returns a real number with the complete time value, including fractions. This book uses `$time` in the examples for brevity, but your testbenches may need to use `$realtime`.

3.8 Conclusion

The new SystemVerilog procedural constructs and task/function features make it easier for you to create testbenches by making the language look more like other programming languages such as C/C++. As a bonus, SystemVerilog has additional HDL constructs such as timing controls, simple thread control, and 4-state logic.

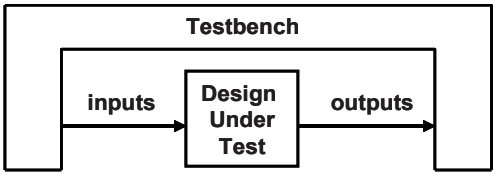
Chapter 4

Connecting the Testbench and Design

There are several steps needed to verify a design: generate stimulus, capture responses, determine correctness, and measure progress. However, first you need the proper testbench, connected to the design as shown in Figure 4-1.

Your testbench wraps around the design, sending in stimulus and capturing the design's response. The testbench forms the "real world" around the design, mimicking the entire environment. For example, a processor model needs to connect to various buses and devices, which are modeled in the testbench as bus functional models. A networking device connects to multiple input and output data streams that are modeled based on standard protocols. A video chip connects to buses that send in commands, and then forms images that are written into memory models. The key concept is that the testbench simulates everything not in the design under test.

Figure 4-1 The testbench – design environment



Your testbench needs a higher-level way to communicate with the design than Verilog's ports and the error-prone pages of connections. You need a robust way to describe the timing so that synchronous signals are always driven and sampled at the correct time and all interactions are free of the race conditions so common to Verilog models.

4.1 Separating the Testbench and Design

In an ideal world, all projects have two separate groups: one to create the design and one to verify it. In the real world, limited budgets may require you to wear both hats. Each team has its own set of specialized skills, such as creating synthesizable RTL code, or figuring out new ways to find bugs in the design. These two groups each read the original design specification and make their own interpretations. The designer has to create code that meets that specification, whereas your job as the verification engineer is to create scenarios where the design does not match its description.

Likewise, your testbench code is in a separate block from design code. In classic Verilog, each goes in a separate module. However, using a module to hold the testbench often causes timing problems around driving and sampling, and so SystemVerilog introduces the program block to separate the testbench, both logically and temporally. For more details, see Section 4.3.

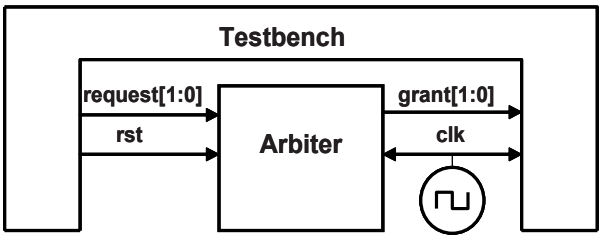
As designs grow in complexity, the connections between the blocks increase. Two RTL blocks may share dozens of signals, which must be listed in the correct order for them to communicate properly. One mismatched or misplaced connection and the design will not work. You can reduce errors by using the connect-by-name syntax, but this more than doubles your typing burden. If it is a subtle error, such as swapping pins that only toggle occasionally, you may not notice the problem for some time. Worse yet is when you add a new signal between two blocks. You have to edit not only the blocks to add the new port but also the higher-level netlists that wire up the devices. Again, one wrong connection at any level and the design stops working. Or worse, the system only fails intermittently!

The solution is the interface, the SystemVerilog construct that represents a bundle of wires, with intelligence such as synchronization, and functional code. An interface can be instantiated like a module but also connected to ports like a signal.

4.1.1 Communication Between the Testbench and DUT

The next few sections show a testbench connected to an arbiter, using individual signals and again using interfaces. Here is a diagram of the top level design including a testbench, arbiter, clock generator, and the signals that connect them as shown in Figure 4-2). This is a trivial design, and so you can concentrate on the SystemVerilog concepts and not get bogged down in the design. At the end of the chapter, an ATM router is shown.

Figure 4-2 Testbench – Arbiter without interfaces



4.1.2 Communication with Ports

The following code fragments show the elements of connecting an RTL block to a testbench. First is the header for the arbiter model. This uses the Verilog-2001 style port declarations, where the type and direction are in the header. Some code has been left out for clarity.

As discussed in Section 2.1.1, SystemVerilog has expanded the classic `reg` type so that you can use it like a `wire` to connect blocks. In recognition of its new capabilities, the `reg` type has the new name of `logic`. The only place where you cannot use a `logic` variable is a net with multiple structural drivers, where you must use a net such as `wire`.

Sample 4.1 Arbiter model using ports

```
module arb_port (output logic [1:0] grant,
                 input  logic [1:0] request,
                 input  logic rst,
                 input  logic clk);
  ...
  always @(posedge clk or posedge rst) begin
    if (rst)
      grant <= 2'b00;
    else
      ...
  end
endmodule
```

The testbench is kept in a module to separate it from the design. Typically, it connects to the design with ports.

Sample 4.2 Testbench using ports

```
module test (input  logic [1:0] grant,
             output logic [1:0] request,
             output logic rst,
             input  logic clk);

  initial begin
    @(posedge clk)       request <= 2'b01;
    $display("@%0t: Drove req=01", $time);
    repeat (2) @(posedge clk);
    if (grant != 2'b01)
      $display("@%0t: a1: grant != 2'b01", $time);
    ...
    $finish;
  end
endmodule
```

The top netlist connects the testbench and DUT, and includes a simple clock generator.

Sample 4.3 Top-level netlist without an interface

```
module top;
  logic [1:0] grant, request;
  bit   clk, rst;
  always #5 clk = ~clk;

  arb_port a1 (grant, request, rst, clk);   // Sample 4.1
  test     t1 (grant, request, rst, clk);   // Sample 4.2
endmodule
```

In Sample 4.3, the netlists are simple, but real designs with hundreds of pins require pages of signal and port declarations. All these connections can be error prone. As a signal moves through several layers of hierarchy, it has to be declared and connected over and over. Worst of all, if you just want to add a new signal, it has to be declared and connected in multiple files. SystemVerilog interfaces can help in each of these cases.

4.2 The Interface Construct

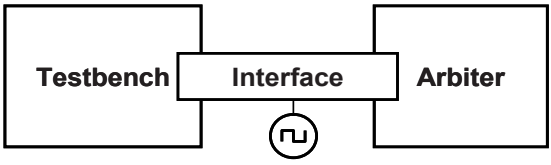
Designs have become so complex that even the communication between blocks may need to be separated out into separate entities. To model this, SystemVerilog uses the interface construct that you can think of as an intelligent bundle of wires. They contain the connectivity, synchronization, and optionally, the functionality of the communication between two or more blocks. They connect design blocks and/or testbenches.

Design-level interfaces are covered in Sutherland (2004). This book concentrates on interfaces that connect design blocks and testbenches.

4.2.1 Using an Interface to Simplify Connections

The first improvement to the arbiter example is to bundle the wires together into an interface. Figure 4-3 shows the testbench and arbiter, communicating using an interface. Note how the interface extends into the two blocks, representing the drivers and receivers that are functionally part of both the test and the DUT. The clock can be part of the interface or a separate port.

Figure 4-3 An interface straddles two modules



The simplest interface is just a bundle of bidirectional signals. Use the `logic` data type so that you can drive the signals from procedural statements.

Sample 4.4 Simple interface for arbiter

```
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  logic rst;
endinterface
```

Sample 4.5 is the device under test, the arbiter, that uses an interface instead of ports.

Sample 4.5 Arbiter using a simple interface

```
module arb (arb_if arbif);
  ...
  always @(posedge arbif.clk or posedge arbif.rst)
    begin
    if (arbif.rst)
      arbif.grant <= 2'b00;
    else
      arbif.grant <= next_grant;
    ...
    end
endmodule
```



The interface instance name, `arbif` in Sample 4.5, should be kept as short as possible as you are going to type it a lot in the design and testbench. You might even consider using a single character, `a`, as long as this is not ambiguous. This book, with its small examples, has short, but not telegraphic names.

Sample 4.6 shows the testbench. You refer to a signal in an interface by making a hierarchical reference using the instance name `arbif.request`. Interface signals should always be driven using nonblocking assignments. This is explained in more detail in Section 4.4.3.

Sample 4.6 Testbench using a simple arbiter interface

```
module test (arb_if arbif);
  ...
  initial begin
    // reset code left out

    @(posedge arbif.clk);
    arbif.request <= 2'b01;
    $display("@%0t: Drove req=01", $time);
    repeat (2) @(posedge arbif.clk);
    if (arbif.grant != 2'b01)
      $display("@%0t: a1: grant != 2'b01", $time);

    $finish;
  end
endmodule : test
```

All these blocks are instantiated and connected in the `top` module.

Sample 4.7 `Top` module using a simple arbiter interface

```
module top;
  bit clk;
  always #5 clk = ~clk;

  arb_if arbif(clk);    // From Sample 4.4
  arb a1 (arbif);       // From Sample 4.5
  test t1(arbif);       // From Sample 4.6
endmodule : top
```

You can see an immediate benefit, even on this small device: the connections become cleaner and less prone to mistakes. If you wanted to put a new signal in an interface, you would just have to add it to the interface definition and the modules that actually used it. You would not have to change any module such as `top` that just pass the interface through. This language feature greatly reduces the chance for wiring errors.



Make sure you declare your interfaces outside of modules and program blocks. If you forget, expect all sorts of trouble. Some compilers may not support defining an interface inside a module. If allowed, the interface would be local to the module and thus not visible to the rest of the design. Sample 4.8 shows the common mistake of including the interface definition right after other include statements.

Sample 4.8 Bad test module includes interface

```
module bad_test(arb_if arbif);
Ôinclude "MyTest.sv"   // Legal include
Ôinclude "arb_if.sv"   // BAD:Interface hidden in module
...
```

4.2.2 Connecting Interfaces and Ports

If you have a Verilog-2001 legacy design with ports that cannot be changed to use an interface, you can just connect the interface's signals to the individual ports. Sample 4.9 connects the original arbiter from Sample 4.1 to the interface in Sample 4.4.

Sample 4.9 Connecting an interface to a module that uses ports

```
module top;
  bit  clk;
  always #5 clk = ~clk;

  arb_if arbif(clk);
  arb_port a1 (.grant   (arbif.grant),     // .port (ifc.signal)
               .request (arbif.request),
               .rst     (arbif.rst),
               .clk     (arbif.clk));
  test t1(arbif);
endmodule : top
```

4.2.3 Grouping Signals in an Interface Using Modports

Sample 4.5 uses a point-to-point connection scheme with no signal directions in the interface. The original netlists using ports had this information that the compiler uses to check for wiring mistakes. The `modport` construct in an interface lets you group signals and specify directions. The MONITOR modport allows you to connect a monitor module.

Sample 4.10 Interface with modports

```
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  logic rst;

  modport TEST (output request, rst,
                input  grant, clk);

  modport DUT (input request, rst, clk,
               output grant);

  modport MONITOR (input request, grant, rst, clk);

endinterface
```

Here are the arbiter model and testbench, with the modport in their port connection list. Note that you put the modport name, `DUT` or `TEST`, after the interface name, `arb_if`. Other than the modport name, these are identical to the previous examples.

Sample 4.11 Arbiter model with interface using modports

```
module arb (arb_if.DUT arbif);
  ...
endmodule
```

Sample 4.12 Testbench with interface using modports

```
module test (arb_if.TEST arbif);
  ...
endmodule
```

The top model does not change from Sample 4.7, as modports are specified in the module header, not when the module is instantiated.

Even though the code didn't change much (except that the interface grew larger), this interface more accurately represents the real design, especially the signal direction.

There are two ways to use these modport names in your design. You can specify them in the program and modules that connect to the interface signals, or you can put them in the top level module that passes the interface into the port list of the program and modules. This book recommends the former, as the modport is an implementation detail that should not clutter the top level module. However, you may want the flexibility to instantiate a module more than once, with each instance connected to a different modport, that is, a different subset of interface signals. In this case, you would need to specify the modport when you instantiate the module, not in the module.

4.2.4 Using Modports with a Bus Design

Not every signal needs to go in every interface. Consider a CPU – memory bus modeled with an `interface`. The CPU is the bus master and drives a subset of the signals, such as `request`, `command`, and `address`. The memory is a slave and receives those signals and drives `ready`. Both master and slave drive `data`. The bus arbiter only looks at `request` and `grant`, and ignores all other signals. So your interface would have three modports for master, slave, and arbiter, plus an optional monitor `modport`.

4.2.5 Creating an Interface Monitor

You can create a bus monitor using the `MONITOR modport`. The following is a trivial monitor for the arbiter. For a real bus, you could decode the commands and print the status: completed, failed, etc.

Sample 4.13 Arbiter model with interface using modports

```
module monitor (arb_if.MONITOR arbif);

  always @(posedge arbif.request[0]) begin
    $display("@%0t: request[0] asserted", $time);
    @(posedge arbif.grant[0]);
    $display("@%0t: grant[0] asserted", $time);
  end

  always @(posedge arbif.request[1]) begin
    $display("@%0t: request[1] asserted", $time);
    @(posedge arbif.grant[1]);
    $display("@%0t: grant[1] asserted", $time);
  end
endmodule
```

4.2.6 Interface Trade-Offs

An interface cannot contain module instances, only instances of other interfaces. There are trade-offs in using interfaces with modports as compared with traditional ports connected with signals. [Au3]

The advantages to using an interface are as follows.

- An interface is ideal for design reuse. When two blocks communicate with a specified protocol using more than two signals, consider using an interface.

If groups of signals are repeated over and over, as in a networking switch, you should additionally use virtual interfaces, as described in Chap. 10.

- The interface takes the jumble of signals that you declare over and over in every module or program and puts it in a central location, reducing the possibility of misconnecting signals.
- To add a new signal, you just have to declare it once in the interface, not in higher-level modules, once again reducing errors.
- Modports allow a module to easily tap a subset of signals from an interface. You can specify signal direction for additional checking.

The disadvantages of using an interface are as follows.

- For point-to-point connections, interfaces with modports are almost as verbose as using ports with lists of signals. Interfaces have the advantage that all the declarations are still in one central location, reducing the chance for making an error.
- You must now use the interface name in addition to the signal name, possibly making the modules more verbose.
- If you are connecting two design blocks with a unique protocol that will not be reused, interfaces may be more work than just wiring together the ports.
- It is difficult to connect two different interfaces. A new interface (`bus_if`) may contain all the signals of an existing one (`arb_if`), plus new signals (address, data, etc.). You may have to break out the individual signals and drive them appropriately.

4.2.7 More Information and Examples

The SystemVerilog LRM specifies many other ways for you to use interfaces. See Sutherland (2004) for more examples of using interfaces for design.

4.3 Stimulus Timing

The timing between the testbench and the design must be carefully orchestrated. At a cycle level, you need to drive and receive the synchronous signals at the proper time in relation to the clock. Drive too late or sample too early, and your testbench is off a cycle. Even within a single time slot (for example, everything that happens at time 100 ns), mixing design and testbench events can cause a race condition, such as when a signal is both read and written at the same time. Do you read the old value, or the one just written? In Verilog, nonblocking assignments help when a test module drives the DUT, but the test could not always be sure it sampled the last value driven by the design. SystemVerilog has several constructs to help you control the timing of the communication.

4.3.1 Controlling Timing of Synchronous Signals with a Clocking Block

An interface block uses a clocking block to specify the timing of synchronous signals relative to the clocks. Any signal in a clocking block is now driven or sampled synchronously, ensuring that your testbench interacts with the signals at the right time. Clocking blocks are mainly used by testbenches but also allow you to create abstract synchronous models.

An interface can contain multiple clocking blocks, one per clock domain, as there is single clock expression in each block. Typical clock expressions are `@(posedge clk)` for a single edge clock and `@(clk)` for a DDR (double data rate) clock.

You can specify a clock skew in the clocking block using the `default` statement, but the default behavior is that input signals are sampled just before the design executes, and the outputs are driven back into the design during the current time slot. The next Section provides more details on the timing between the design and testbench.

Once you have defined a clocking block, your testbench can wait for the clocking expression with `@arbif.cb` rather than having to spell out the exact clock and edge. Now if you change the clock or edge in the clocking block, you do not have to change your testbench.

Sample 4.14 is similar to Sample 4.10 except that the `TEST` modport now treats `request` and `grant` as synchronous signals. The clocking block `cb` declares that the signals are active on the positive edge of the clock. The signal directions are relative to the `modport` where they are used. So `request` is an output in the `TEST` modport, and `grant` is an input.

Sample 4.14 Interface with a clocking block

```
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  logic rst;

  clocking cb @(posedge clk);    // Declare cb
    output request;
    input grant;
  endclocking

  modport TEST (clocking cb,     // Use cb
                output rst);

  modport DUT (input request, rst, output grant);
endinterface

// Trivial test, see Sample 4.20 for a better one
module test(arb_if.TEST arbif);
  initial begin
    arbif.cb.request <= 0;
    @arbif.cb;
    $display("@%0t: Grant = %b", $time, arbif.cb.grant);
  end
endmodule
```

4.3.2 Logic vs. Wire in an Interface

This book recommends declaring the signals in your interface as `logic`, while the VMM has a rule that says to use a `wire`. The difference is ease-of-use vs. reusability.

If your testbench drives an asynchronous signal in an interface with a procedural assignment, the signal must be a `logic` type. A `wire` can only be driven with a continuous assignment statement. Signals in a clocking block are always synchronous and can be declared as `logic` or `wire`. Sample 4.15 shows how the `logic` signal can be driven directly, whereas the `wire` requires additional code.

Sample 4.15 Interface with a clocking block

```
interface asynch_if();
  logic l;
  wire w;

endinterface

module test(asynch_if ifc);
  logic local_wire;
  assign ifc.w <= local_wire;

  initial begin
    ifc.l <= 0;       // Drive asych logic directly ...
    local_wire <= 1; // but drive wire through assign
    ...
  end
endmodule
```

Another reason to use `logic` for interface signals is that the compiler will give an error if you unintentionally use multiple structural drivers.

The VMM takes a more long-term approach. Take the case where you have created test code that works well on the current project and is later used in a new design. What if your interface with all its `logic` signals is connected such that now a signal has multiple structural drivers? The engineers will have to change that `logic` to a `wire`, and, if the signal does not go through a clocking block, change the procedural assignment statements. Now there are two versions of the interface, and existing tests must be modified before they can be reused. Rewriting good code goes against the VMM principles.

4.3.3 Timing Problems in Verilog

Your testbench needs to be separate from the design, not just logically but also temporally. Consider how a hardware tester interacts with a chip for synchronous signals. In a real hardware design, the DUT's storage elements latch their inputs from the tester at the active clock edge. These values propagate through the storage outputs, and then the logic clouds to the inputs of the next storage elements. The time from the input of the first storage to the next must be less than a clock cycle. So a hardware tester needs to drive the chip's input at the clock edge, and read the outputs just before the following edge.

A testbench has to mimic this tester behavior. It should drive on or after the active clock edge, and should sample as late as possible as allowed by the protocol timing specification, just before the active clock edge.

If the DUT and testbench are made of Verilog modules only, this outcome is nearly impossible to achieve. If the testbench drives the DUT at the clock edge, there could

be race conditions. What if the clock propagates to some DUT inputs before the TB stimulus, but is a little later to other inputs? From the outside, the clock edges all arrive at the same simulation time, but in the design, some inputs get the value driven during the last cycle, whereas other inputs get values from the current cycle.

One way around this problem is to add small delays to the system, such as `#0`. This forces the thread of Verilog code to stop and be rescheduled after all other code. Invariably though, a large design has several sections that all want to execute last. Whose `#0` wins out? It could vary from run to run and be unpredictable between simulators. Multiple threads using `#0` delays cause indeterministic behavior. Avoid using `#0` as it will make your code unstable and not portable.

The next solution is to use a larger delay, `#1`. RTL code has no timing, other than clock edges, and so one time unit after the clock, the logic has settled. However, what if one module uses a time precision of 1 ns, whereas another used a resolution of just 10 ps? Does that `#1` mean 1 ns, 10 ps, or something else? You want to drive as soon as possible after the clock cycle with the active clock edge, but not during that time, and before anything else can happen. Worse yet, your DUT may contain a mix of RTL code with no delays and gate code with delays. Just as you should avoid using `#0`, stay away from `#1` delays to fix timing problems.

4.3.4 Testbench – Design Race Condition

Sample 4.16 shows a potential race condition between the testbench and design. The race condition occurs when the test drives the `start` signal and then the other ports. The memory is waiting on the `start` signal and could wake up immediately, whereas the `write`, `addr`, and `data` signals still have their old values. You could delay all these signals slightly by using nonblocking assignments, as recommended by Cummings (2000), but remember that the testbench and the design are both using these assignments. It is still possible to get a race condition between the testbench and design.

Sampling the design outputs has a similar problem. You want to grab the values at the last possible moment, just before the active clock edge. Perhaps you know the next clock edge is at 100 ns. You can't sample right at the clock edge at 100 ns, as some design values may have already changed. You should sample at `Tsetup` just before the clock edge.

Sample 4.16 Race condition between testbench and design

```
module memory(input wire start, write,
              input wire [7:0] addr,
              inout wire [7:0] data);
  logic [7:0] mem[256];
  always @(posedge start) begin
    if (write)
      mem[addr] <= data;
    ...
  end
endmodule

module test(output logic start, write,
            output logic [7:0] addr, data);
  initial begin
    start = 0;            // Initialize signals
    write = 0;
    #10;                  // Short delay
    addr = 8'h42;         // Start first command
    data = 8'h5a;
    start = 1;
    write = 1;
    ...
  end
endmodule
```

4.3.5 The Program Block and Timing Regions

The root of the problem is the mixing of design and testbench events during the same time slot, though even in pure RTL the same problem can happen1[1]. What if there were a way you could separate these events temporally, just as you separated the code? At 100 ns, your testbench could sample the design outputs before the clock has had a chance to change and any design activity has occurred. By definition, these values would be the last possible ones from the previous time slot. Then, after all the design events are done, your testbench would start.

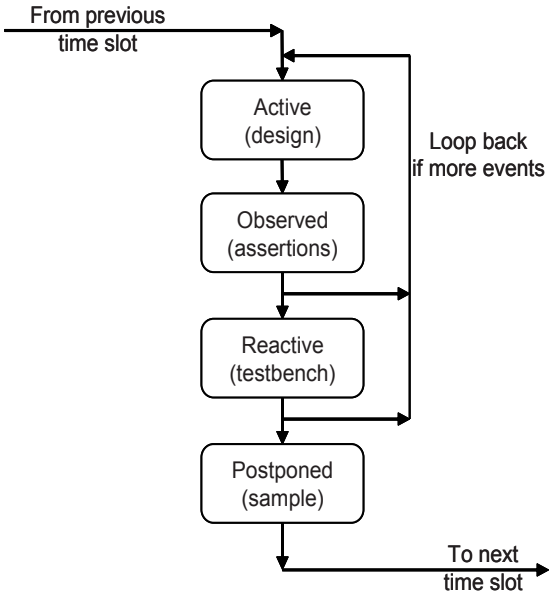
How does SystemVerilog know to schedule the testbench events separately from the design events? In SystemVerilog, your testbench code is in a program block, which is similar to a module in that it can contain code and variables and be instantiated in other modules. However, a program cannot have any hierarchy such as instances of modules, interfaces, or other programs.

A new division of the time slot was introduced in SystemVerilog as shown in Figure 4-4. In Verilog, most events executed in the Active region. There are dozens of other regions for nonblocking assignments, PLI execution, etc., but they can be ignored for

[1]Good coding guidelines such as proper use of nonblocking assignments can reduce these race conditions, but improperly coded assignments have the habit of creeping in. Bugs happen, even in testbenches.

the purposes of this book. See the LRM and Cummings and Salz (2006) for more details on the SystemVerilog event regions.

Figure 4-4 Main regions inside a SystemVerilog time step



First to execute during a time slot is the Active region, where design events run. These include your RTL and gate code plus the clock generator. The second region is the Observed region, where assertions are evaluated. Following that is the Reactive region where the testbench executes. Note that time does not strictly flow forwards – events in the Observed and Reactive regions can trigger further design events in the Active region in the current cycle. Lastly is the Postponed region, which samples signals at the end of the time slot, in the read-only period, after design activity has completed as shown in Table 4-1.

Table 4-1 Primary SystemVerilog scheduling regions

Name	Activity
Active	Simulation of design code in modules
Observed	Evaluation of SystemVerilog Assertions
Reactive	Execution of testbench code in programs
Postponed	Sampling design signals for testbench input

Sample 4.17 shows part of the testbench code for the arbiter. Note that the statement `@arbif.cb` waits for the active edge of the clocking block, `@(posedge clk)`, as shown in Sample 4.14.

Sample 4.17 Testbench using interface with clocking block

```
program automatic test (arb_if.TEST arbif);
  ...
  initial begin
    arbif.cb.request <= 2'b01;
    $display("@%0t: Drove req=01", $time);
    repeat (2) @arbif.cb;
    if (arbif.cb.grant != 2'b01)
      $display("@%0t: a1: grant != 2'b01", $time);
    end

endprogram : test
```

Section 4.4 explains more about the driving and sampling of interface signals.



Your test should be contained in a single program. You should use OOP to build a dynamic, hierarchical testbench from objects instead of modules. A simulation may have multiple program blocks if you are using code from other people or combining several tests.



As discussed in Section 3.6.1, you should always declare your program block as `automatic` so that it behaves more like the routines in stack-based languages you may have worked with, such as C.

4.3.6 The End of Simulation

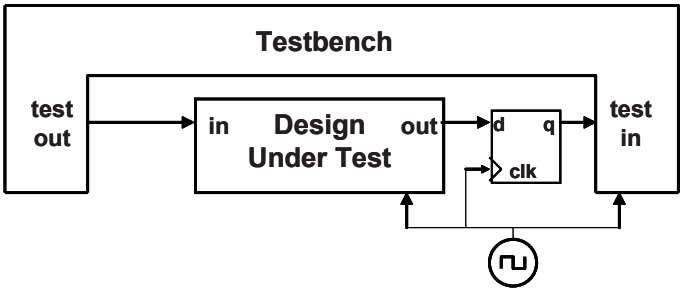
In Verilog, simulation continues while there are scheduled events, or until a `$finish` is executed. SystemVerilog adds an additional way to end simulation. A program block is treated as if it contains a test. If there is only a single program block, simulation ends when you complete the last statement in every `initial`-block, as this is considered the end of the test. Simulation ends even if there are threads still running in the program or modules. As a result, you don't have to shut down every monitor and driver when a test is done.

If there are several program blocks, simulation ends when the last program completes. This way simulation ends when the last test completes. You can terminate any program block early by executing `$exit`. Of course you can still use `$finish` to end simulation.

4.3.7 Specifying Delays Between the Design and Testbench

The default timing of the clocking block is to sample inputs with a delay of `#1step` and to drive the outputs with a delay of `#0`. The `1step` delay specifies that signals are sampled in the Postponed region of the previous time slot, before any design activity. So you get the output values just before the clock changes. The testbench outputs are synchronous by virtue of the clocking block, and so they flow directly into the design. The program block, running in the Reactive region, retriggers the Active region during the same time slot. If you have a design background, you can remember this by imagining that the clocking block inserts a synchronizer between the design and testbench as shown in Figure 4-5.

Figure 4-5 A clocking block synchronizes the DUT and testbench



4.4 Interface Driving and Sampling

Your testbench needs to drive and sample signals from the design, primarily through interfaces with clocking blocks. The next Section uses the arbiter interface from Sample 4.14 and the top-level module from Sample 4.9.

Asynchronous signals such as `rst` pass through the interface with no delays. The signals in the clocking block get synchronized as shown in the sections below.

4.4.1 Interface Synchronization

You can use the Verilog `@` and `wait` constructs to synchronize with the signals in a testbench. The following code does not do anything useful except to show the various constructs.

Sample 4.18 Signal synchronization

```
program automatic test(bus_if.TB bus);
  initial begin
    @bus.cb;                  // Continue on active edge
                              // in clocking block
    repeat (3) @bus.cb;       // Wait for 3 active edges
    @bus.cb.grant;            // Continue on any edge
    @(posedge bus.cb.grant); // Continue on posedge
    @(negedge bus.cb.grant); // Continue on negedge
    wait (bus.cb.grant==1);  // Wait for expression
                              // No delay if already true
    @(posedge bus.cb.grant or
      negedge bus.rst);       // Wait for several signals
  end
endprogram
```

4.4.2 Interface Signal Sample

When you read a signal from a clocking block, you get the sample from just before the last clock edge, i.e., from the Postponed region. The following code shows a program block that reads the synchronous grant signal from the DUT. The `arb` module drives `grant` to 1 and 2 in the middle of a cycle, and then to 3 exactly at the clock edge.

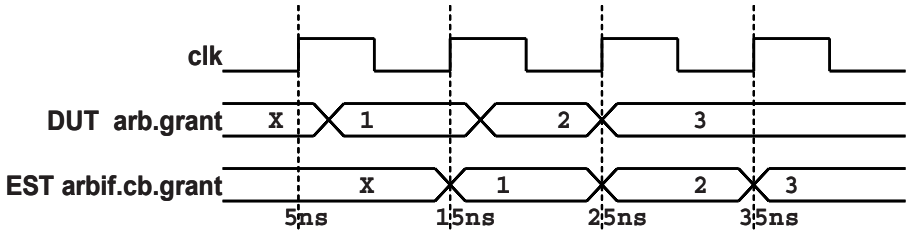
Sample 4.19 Synchronous interface sample and drive from module

```
Ôtimescale 1ns/1ns
program test(arb_if.TEST arbif);
  initial begin
    $monitor("@%0t: grant=%h", $time, arbif.cb.grant);
    #50ns $display("End of test");
  end
endprogram

module arb(arb_if.DUT arbif);
  initial begin
    #7  arbif.grant = 1;  // @ 7ns
    #10 arbif.grant = 2;  // @ 17ns
    #8  arbif.grant = 3;  // @ 25ns
  end
endmodule
```

The waveforms in Figure 4-6 show that in the program, `arbif.cb.grant` gets the value from just before the clock edge. When the interface input changes right at a clock edge, 25 ns, that value does not propagate to the testbench for another cycle, 35 ns.

Figure 4-6 Sampling a synchronous interface



4.4.3 Interface Signal Drive

Here is an abbreviated version of the arbiter test program, which uses the arbiter interface in Sample 4.14.

Sample 4.20 Testbench using interface with clocking block

```
program automatic test (arb_if.TEST arbif);

  initial begin
    arbif.cb.request <= 2'b01;
    $display("@%0t: Drove req=01", $time);
    repeat (2) @arbif.cb;
    if (arbif.cb.grant != 2'b01)
      $display("@%0t: grant != 2'b01", $time);
    end

endprogram : test
```



When using modports with clocking blocks, a synchronous interface signal such as `request` must be prefixed with both the interface name, `arbif`, and the clocking block name, `cb`. So in Sample 4.20, `arbif.cb.request` is legal, but `arbif.request` is not. This is the most common coding mistake with interfaces and clocking blocks.

4.4.4 Driving Interface Signals Through a Clocking Block

You should always drive interface signals in a clocking block with a synchronous drive using the `<=` operator.[2] This is because the design signal does not change immediately after your assignment – remember that your testbench executes in the Reactive region while design code is in the Active region. If your testbench drives `arbif.cb.request` at 100 ns, the same time as `arbif.cb` (which is `@(posedge clk)` according to the clocking block), `request` changes in the design at 100 ns.

[2]Yes, this does look like a nonblocking assignment, but the LRM insists that this is something different.

However, if your testbench tries to drive `arbif.cb.request` at time 101 ns, between clock edges, the change does not propagate until the next clock edge. In this way, your drives are always synchronous. In Sample 4.19, `arbif.grant` is driven by a module and can use a blocking assignment.

If the testbench drives the synchronous interface signal at the active edge of the clock, the value propagates immediately to the design. This is because the default output delay is `#0` for a clocking block. If the testbench drives the output just after the active edge, the value is not seen in the design until the next active edge of the clock.

Sample 4.21 Interface signal drive

```
busif.cb.request <= 1;       // Synchronous drive
busif.cb.cmd <= cmd_buf;     // Synchronous drive
```

Sample 4.22 shows what happens if you drive a synchronous interface signal at various points during a clock cycle. This uses the interface from Sample 4.14 and the top module and clock generator from Sample 4.9.

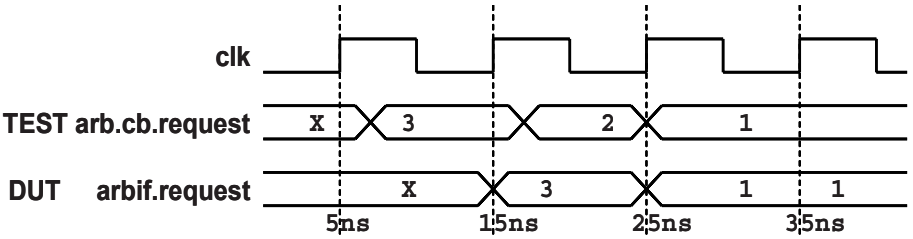
Sample 4.22 Driving a synchronous interface

```
program test(arb_if.TEST arbif);
  initial begin
    # 7 arbif.cb.request <= 3;  // @ 7ns
    #10 arbif.cb.request <= 2;  // @ 17ns
    # 8 arbif.cb.request <= 1;  // @ 25ns
    #15 finish;
  end
endprogram

module arb(arb_if.DUT arbif);
  initial
    $monitor("@%0t: req=%h", $time, arbif.request);
endmodule
```

Note that in Figure 4-7, the value 3, driven in the middle of the second cycle, is seen by the DUT at the start of the third cycle. The value 2 is driven in the middle of the third cycle, and is never seen by the DUT as the testbench drives a 1 at the end of the third cycle.

Figure 4-7 Driving a synchronous interface



Driving clocking block signals asynchronously can lead to dropped values. Instead, drive at the clock edge by using a cycle delay prefix on your drives, as shown in Sample 4.23.

Sample 4.23 Interface signal drive

```
##2 arbif.cb.request <= 0;  // Wait 2 cycles then assign
##3;         // Illegal - must be used with an assignment
```

If you want to wait for two clock cycles before driving a signal, you can either use "`repeat (2) @bus.cb;`" or use the cycle delay `##2`. This latter delay only works as a prefix to a drive of a signal in a clocking block, as it needs to know which clock to use for the delay. (`##3;` does work if you have a default clocking block for your program or module, but this book only recommends putting a clocking block in an interface.)

4.4.5 Bidirectional Signals in the Interface

In Verilog-1995, if you want to drive a bidirectional signal such as a port from procedural code, you need a continuous assignment to connect the `reg` to the `wire`. In SystemVerilog, synchronous bidirectional signals in interfaces are easier to use as the continuous assignment is added for you. When you write to the net from a program, SystemVerilog actually writes to a temporary variable that drives the net. Your program reads directly from the wire, seeing the value that is resolved from all the drivers. Design code in a module still uses the classic register plus continuous assignment statement.

Sample 4.24 Bidirectional signals in a program and interface

```
interface master_if (input bit clk);
  wire [7:0] data;  // Bidirectional signal

  clocking cb @(posedge clk);
    inout data;
  endclocking

  modport TEST (clocking cb);
endinterface


program test(master_if.TEST mif);

  initial begin
    mif.cb.data <= 'z;          // Tri-state the bus
    @mif.cb;
    $displayh(mif.cb.data);     // Read from the bus
    @mif.cb;
    mif.cb.data <= 7Õh5a;       // Drive the bus
    @mif.cb;
    mif.cb.data <= 'z;          // Release the bus
  end

endprogram
```

The SystemVerilog LRM is not clear on driving an asynchronous bidirectional signal using an interface. Two possible solutions are to use a cross-module reference and continuous assignment or to use a virtual interface as shown in Chap. 10.

4.4.6 Why are `always` Blocks Not Allowed in a Program?

In SystemVerilog, you can put `initial` blocks in a program, but not `always` blocks. This may seem odd if you are used to Verilog modules, but there are several reasons. SystemVerilog programs are closer to a program in C, with one (or more) entry points, than Verilog's many small blocks of concurrently executing hardware. In a design, an `always` block might trigger on every positive edge of a clock from the start of simulation. In contrast, a testbench has the steps of initialization, stimulate and respond to the design, and then wrap up simulation. An `always` block that runs continuously would not work.

When the last `initial` block completes in the program, simulation implicitly ends just as if you had executed `$finish`. If you had an `always` block, it would never stop, and so you would have to explicitly call `$exit` to signal that the program block completed.

But don't despair. If you really need an `always` block, you can use `initial forever` to accomplish the same thing.

4.4.7 The Clock Generator

Now that you have seen the program block, you may wonder if the clock generator should be in a module. The clock is more closely tied to the design than the testbench, and so the clock generator should remain in a module. As you refine the design, you create clock trees, and you have to carefully control the skews as the clocks enter the system and propagate through the blocks.

The testbench is much less picky. It just wants a clock edge to know when to drive and sample signals. Functional verification is concerned with providing the right values at the right cycle, not with fractional nanosecond delays and relative clock skews.

Sample 4.25 Bad clock generator in program block

```
program bad_generator (output bit clk, out_sig);
  initial
    forever #5 clk <= ~clk ;

  initial
    forever @(posedge clk)
      out_sig <= ~out_sig;
endprogram
```

The program block is not the place to put a clock generator. Sample 4.25 tries to put the generator in a program block but just causes a race condition. The `clk` and `out_sig` signals both propagate from the Reactive region to the design in the Active region and could cause a race condition depending on which one arrived first.



Avoid race conditions by always putting the clock generator in a module. If you want to randomize the generator's properties, create a class with random variables for skew, frequency, and other characteristics, as shown in Chap. 6. You can use this class in the generator module, or in the testbench.

Sample 4.26 shows a good clock generator in a module. It deliberately avoids an edge at time 0 to avoid race conditions. All clock edges are generated with a blocking assignment so as to trigger events during the Active region. If you really need to generate a clock edge at time 0, use a nonblocking assignment to set the initial value so all clock sensitive logic such as `always` blocks will have started before the clock changes value.

Sample 4.26 Good clock generator in module

```
module clock_generator (output bit clk);
  initial
    forever #5 clk = ~clk; // Generate edges after time 0
endmodule
```



Lastly, don't try to verify the low-level timing with functional verification. The testbenches described in this book check the behavior of the DUT but not the timing, which is better done with a static timing analysis tool.

4.5 Connecting It All Together

Now you have a design described in a module, a testbench in a program block, and interfaces that connect them together. Here is the top-level module that instantiates and connects all the pieces.

Sample 4.27 `Top` module using a simple arbiter interface

```
module top;
  bit  clk;
  always #5 clk = ~clk;

  arb_if arbif(.*);
  arb a1 (.*);
  test t1(.*);
endmodule : top
```

This is almost identical to Sample 4.7. It uses a shortcut notation `.*` (implicit port connection) that automatically connects module instance ports to signals at the current level if they have the same name and data type.

4.5.1 An Interface in a Port List Must be Connected

SystemVerilog compiler won't let you compile a single module or program that uses an interface in the port list. Why not? After all, a module or program with ports made of individual signals can be compiled without being instantiated, as shown in Sample 4.28.

Sample 4.28 Module with just port connections

```
module uses_a_port(inout bit not_connected);
  ...
endmodule
```

The compiler creates wires and connects them to the dangling signals. However, a module or program with an interface in its port list must be connected to an instance of the interface.

Sample 4.29 Module with an interface

```
// This will not compile without interface declaration
module uses_an_interface(arb_ifc.DUT ifc);
  initial ifc.grant = 0;
endmodule
```

For Sample 4.29, the compiler is not able to build a complex interface with the necessary modports. If you have a program block using clocking blocks in an interface, the compiler has an even more difficult time. Even if you are just looking to wring out syntax bugs, you must complete the connections. This can be done as shown in Sample 4.30.

Sample 4.30 Top module connecting DUT and interface

```
module top;
  bit clk;
  always #10 clk = !clk;

  arb_ifc ifc(clk);           // Interface with clocking block
  uses_an_interface u1(ifc);  //  that is needed to compile this
endmodule
```

4.6 Top-Level Scope

Sometimes you need to create things in your simulation that are outside of a program or module so that they are seen by all parts of the simulation. In Verilog, only macros extend across module boundaries, and are often used for creating global constants. SystemVerilog introduces the *compilation unit,* which is a group of source files that are compiled together. The scope outside the boundaries of any `module`, `macromodule`, `interface`, `program`, `package`, or `primitive` is known as the *compilation-unit scope*, also referred to as `$unit`. Anything such as a `parameter` defined in this scope is similar to a global because it can be seen by all lower-level blocks. However, it is not truly global as the `parameter` cannot be seen during compilation of other files.

This leads to some confusion. Some tools, such as Synopsys VCS, compile all the SystemVerilog code together, and so `$unit` is global. On the other hand, Synopsys Design Compiler compiles a single module or group of modules at a time, and so `$unit` may be just the contents of one or a few files. Tools from other vendors may compile all files or just a subset at once. As a result, `$unit` is not portable.

This book calls the scope outside blocks the "top-level scope." You can define variables, parameters, data types, and even routines in this space. Sample 4.31 declares a top-level parameter, `TIMEOUT`, that can be used anywhere in the hierarchy. This example also has a `const` string that holds an error message. You can declare top-level constants either way.

Sample 4.31 Top-level scope for arbiter design

```
// root.sv
`timescale 1ns/1ns
parameter int TIMEOUT = 1_000_000;
const string time_out_msg = "ERROR: Time out";
module top;
  test t1();
endmodule

program automatic test;
  ...
  initial begin
    #TIMEOUT;
    $display("%s", time_out_msg);
    $finish;
    end
endprogram
```

The instance name `$root` allows you to unambiguously refer to names in the system, starting with the top-level scope. In this respect, `$root` is similar to "`/`" in the Unix file system. For tools such as VCS that compile all files at once, `$root` and `$unit` are equivalent. The name `$root` also solves an old Verilog problem. When your code refers to a name in another module, such as `i1.var`, the compiler first looks in the local scope, then looks up to the next higher scope, and so on until it reaches the top. You may have wanted to use `i1.var` in the top module, but an instance named `i1` in an intermediate scope may have sidetracked the search, giving you the wrong variable. You use `$root` to make unambiguous cross module references by specifying the absolute path.

Sample 4.32 shows a program that is instantiated in a module that is explicitly instantiated in the top-level scope. The program can use a relative or absolute reference to the `clk` signal in the module. Note that if the module were implicitly instantiated, that is, if you took out the line `top t1();`, the absolute reference in the program would change to `$root.top.clk`. Use explicit instantiation of the top module if you plan on making cross-module references. You may want to use a macro to hold the hierarchical path so that when the path changes, you only have to change one piece of code.

Sample 4.32 Cross-module references with `$root`

```
`timescale 1ns/1ns
parameter TIMEOUT = 1_000_000;
top t1();       // Explicitly instantiate top-level module

module top;
  bit clk;
  test t1(.*);
endmodule

Ôdefine TOP $root.t1
program automatic test;
  ...
  initial begin
    // Absolute reference
    $display("clk=%b", $root.t1.clk);
    $display("clk=%b", ÔTOP.clk);    // With macro

    // Relative reference
    $display("clk=%b", t1.clk);
    end
endprogram
```

4.7 Program – Module Interactions

The program block can read and write all signals in modules, and can call routines in modules, but a module has no visibility into a program. This is because your testbench needs to see and control the design, but the design should not depend on anything in the testbench.



A program can call a routine in a module to perform various actions. The routine can set values on internal signals, also known as "back-door load." Next, because the current SystemVerilog standard does not define how to force signals from a program block, you need to write a task in the design to do the force, and then call it from the program.

Lastly, it is a good practice for your testbench to use a function to get information from the DUT. Reading signal values can work most of the time, but if the design code changes, your testbench may interpret the values incorrectly. A function in the module can encapsulate the communication between the two and make it easier for your testbench to stay synchronized with the design.

4.8 SystemVerilog Assertions

You can create temporal assertions about signals in your design using SystemVerilog Assertions (SVA). Assertions are instantiated similarly to other design blocks and are active for the entire simulation. The simulator keeps track of what assertions have triggered, and so you can gather functional coverage data on them.

4.8.1 Immediate Assertions

Your testbench procedural code can check the values of design signals and testbench variables and take action if there is a problem. For example, if you have asserted the bus request, you expect that grant will be asserted two cycles later. You could use an `if`-statement.

Sample 4.33 Checking a signal with an `if`-statement

```
bus.cb.request <= 1;
repeat (2) @bus.cb;
if (bus.cb.grant != 2Õb01)
  $display("Error, grant != 1");
// rest of the test
```

An assertion is more compact than an `if`-statement. However, note that the logic is reversed compared to the `if`-statement above. You want the expression inside the parentheses to be true; otherwise, print an error.

Sample 4.34 Simple immediate assertion

```
bus.cb.request <= 1;
repeat (2) @bus.cb;
a1: assert (bus.cb.grant == 2Õb01);
// rest of the test
```

If the `grant` signal is asserted correctly, the test continues. If the signal does not have the expected value, the simulator produces a message similar to the following.

Sample 4.35 Error from failed immediate assertion

```
"test.sv", 7: top.t1.a1: started at 55ns failed at 55ns
Offending '(bus.cb.grant == 2Õb1)Õ
```

This says that on line 7 of the file `test.sv`, the assertion `top.t1.a1` started at 55 ns to check the signal `bus.cb.grant`, but failed immediately.



You may be tempted to use the full SystemVerilog Assertion syntax to check an elaborate sequence over a range of time, but use carefully. Assertions are declarative code, and execute very differently than the surrounding procedural code. In just a few lines of assertions, you can verify temporal relations; the equivalent procedural code would be far more complicated and verbose.

4.8.2 Customizing the Assertion Actions

An immediate assertion has optional then- and else-clauses. If you want to augment the default message, you can add your own.

Sample 4.36 Creating a custom error message in an immediate assertion

```
a1: assert (bus.cb.grant == 2'b01)
else $error("Grant not asserted");
```

If `grant` does not have the expected value, you'll see an error message.

Sample 4.37 Error from failed immediate assertion

```
 "test.sv", 7: top.t1.a1: started at 55ns failed at 55ns
Offending '(bus.cb.grant == 2'b1)'
Error: "test.sv", 7: top.t1.a1: at time 55 ns
Grant not asserted
```

SystemVerilog has four functions to print messages: `$info`, `$warning`, `$error`, and `$fatal`. These are allowed only inside an assertion, not in procedural code, though future versions of SystemVerilog may allow this.

You can use the then-clause to record when an assertion completed successfully.

Sample 4.38 Creating a custom error message

```
a1: assert (bus.cb.grant == 2'b01)
  grants_received++;        // Another succesful result
else
  $error("Grant not asserted");
```

4.8.3 Concurrent Assertions

The other type of assertion is the concurrent assertion that you can think of as a small model that runs continuously, checking the values of signals for the entire simulation. You need to specify a sampling clock in the assertion. Here is a small assertion to check that the arbiter request signal does not have X or Z values except during reset.

Sample 4.39 Concurrent assertion to check for X/Z

```
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  logic rst;

  property request_2state;
    @(posedge clk) disable iff (rst)
    $isunknown(request) == 0;        // Make sure no Z or X found
  endproperty
  assert_request_2state: assert property (request_2state);

endinterface
```

4.8.4 Exploring Assertions

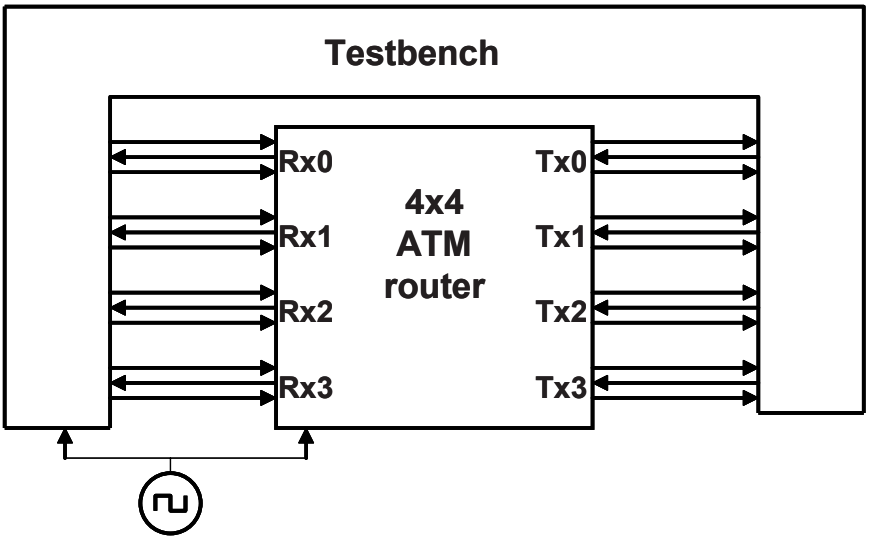
There are many other uses for assertions. For example, you can put assertions in an interface. Now your interface not only transmits signal values but also checks the protocol.

This Section provides a brief introduction to SystemVerilog Assertions. For more information, see Vijayaraghhavan and Ramanathan (2005) and Haque et al. (2006).

4.9 The Four-Port ATM Router

The arbiter example is a good introduction to interfaces, but real designs have more than a single input and output. This Section discusses a four-port ATM (Asynchronous Transfer Mode) router, shown in Figure 4-8.

Figure 4-8 Testbench – ATM router diagram without interfaces



4.9.1 ATM Router with Ports

The following code fragments show the tangle of wires you would have to endure to connect an RTL block to a testbench. First is the header for the ATM router model. This uses the Verilog-1995 style port declarations, where the type and direction are separate from the header.

The actual code for the router is crowded out by nearly a page of port declarations.

Sample 4.40 ATM router model header without an interface

```
module atm_router(
  // 4 x Level 1 Utopia ATM layer Rx Interfaces
     Rx_clk_0,  Rx_clk_1,  Rx_clk_2,  Rx_clk_3,
     Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
     Rx_soc_0,  Rx_soc_1,  Rx_soc_2,  Rx_soc_3,
     Rx_en_0,   Rx_en_1,   Rx_en_2,   Rx_en_3,
     Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,

  // 4 x Level 1 Utopia ATM layer Tx Interfaces
     Tx_clk_0,   Tx_clk_1,   Tx_clk_2,   Tx_clk_3,
     Tx_data_0,  Tx_data_1,  Tx_data_2,  Tx_data_3,
     Tx_soc_0,   Tx_soc_1,   Tx_soc_2,   Tx_soc_3,
     Tx_en_0,    Tx_en_1,    Tx_en_2,    Tx_en_3,
     Tx_clav_0,  Tx_clav_1,  Tx_clav_2,  Tx_clav_3,

  // Miscellaneous control interfaces
     rst, clk);

// 4 x Level 1 Utopia Rx Interfaces
  output       Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3;
  input [7:0]  Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3;
  input        Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3;
  output       Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3;
  input        Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3;

// 4 x Level 1 Utopia Tx Interfaces
  output       Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3;
  output [7:0] Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3;
  output       Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3;
  output       Tx_en_0,  Tx_en_1,  Tx_en_2,  Tx_en_3;
  input        Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3;

// Miscellaneous control interfaces
  input        rst, clk;
  ...[3]
endmodule
```

4.9.2 ATM Top-Level Netlist with Ports

Shown next is the top-level netlist.

[3]So what goes in the "..."? See Sutherland (2006) for more information and examples of using interfaces in modules.

Sample 4.41 Top-level netlist without an interface

```
module top;
  bit clk;
  always #5 clk = !clk;
  wire Rx_clk_0,  Rx_clk_1,  Rx_clk_2,  Rx_clk_3,
       Rx_soc_0,  Rx_soc_1,  Rx_soc_2,  Rx_soc_3,
       Rx_en_0,   Rx_en_1,   Rx_en_2,   Rx_en_3,
       Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,
       Tx_clk_0,  Tx_clk_1,  Tx_clk_2,  Tx_clk_3,
       Tx_soc_0,  Tx_soc_1,  Tx_soc_2,  Tx_soc_3,
       Tx_en_0,   Tx_en_1,   Tx_en_2,   Tx_en_3,
       Tx_clav_0, Tx_clav_1, Tx_clav_2, Tx_clav_3, rst;

  wire [7:0] Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
             Tx_data_0, Tx_data_1, Tx_data_2, Tx_data_3;

  atm_router a1(Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
                Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3,
                Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
                Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3,
                Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3,
                Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
                Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3,
                Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
                Tx_en_0,  Tx_en_1,  Tx_en_2,  Tx_en_3,
                Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3,
                rst, clk);

  test      t1 (Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
                Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3,
                Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
                Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3,
                Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3,
                Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
                Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3,
                Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
                Tx_en_0,  Tx_en_1,  Tx_en_2,  Tx_en_3,
                Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3,
                rst, clk);
endmodule
```

Sample 4.42 shows the top of the testbench module. Once again, note that the ports and wires take up the majority of the netlist.

Sample 4.42 Verilog-1995 testbench using ports

```
module test(
    // 4 x Level 1 Utopia ATM layer Rx Interfaces
    Rx_clk_0,  Rx_clk_1,  Rx_clk_2,  Rx_clk_3,
    Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
    Rx_soc_0,  Rx_soc_1,  Rx_soc_2,  Rx_soc_3,
    Rx_en_0,   Rx_en_1,  Rx_en_2,   Rx_en_3,
    Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,

    // 4 x Level 1 Utopia ATM layer Tx Interfaces
    Tx_clk_0,  Tx_clk_1,  Tx_clk_2,  Tx_clk_3,
    Tx_data_0, Tx_data_1, Tx_data_2, Tx_data_3,
    Tx_soc_0,  Tx_soc_1,  Tx_soc_2,  Tx_soc_3,
    Tx_en_0,   Tx_en_1,  Tx_en_2,   Tx_en_3,
    Tx_clav_0, Tx_clav_1, Tx_clav_2, Tx_clav_3,

    // Miscellaneous control interfaces
    rst, clk);

// 4 x Level 1 Utopia Rx Interfaces
  input        Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3;
  output [7:0] Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3;
  reg  [7:0]   Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3;
  output       Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3;
  reg          Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3;
  input        Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3;
  output       Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3;
  reg          Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3;

// 4 x Level 1 Utopia Tx Interfaces
  input       Tx_clk_0,  Tx_clk_1, Tx_clk_2, Tx_clk_3;
  input [7:0] Tx_data_0, Tx_data_1,Tx_data_2,Tx_data_3;
  input       Tx_soc_0,  Tx_soc_1, Tx_soc_2, Tx_soc_3;
  input       Tx_en_0,  Tx_en_1,  Tx_en_2,   Tx_en_3;
  output      Tx_clav_0, Tx_clav_1,Tx_clav_2,Tx_clav_3;
  reg         Tx_clav_0, Tx_clav_1,Tx_clav_2,Tx_clav_3;

// Miscellaneous control interfaces
  output       rst;
  reg          rst;
  input        clk;

  initial begin
    // Reset the device
    rst <= 1;
    Rx_data_0 <= 0;
    ...
    end
```

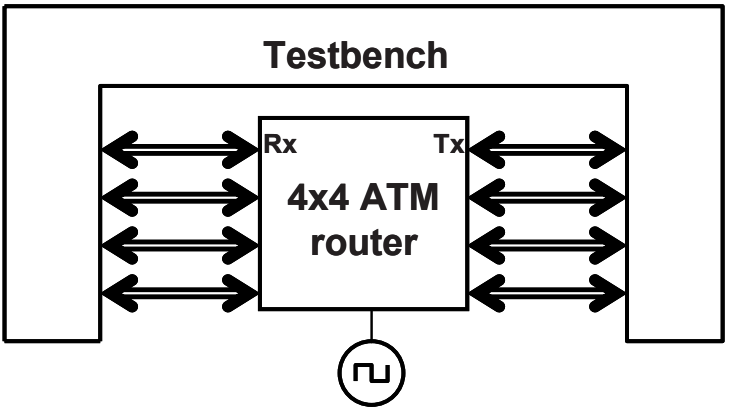
```
endmodule
```

You just saw three pages of code, and it was all just connectivity – no testbench, no design! Interfaces provide a better way to organize all this information and eliminate the repetitive parts that are so error prone.

4.9.3 Using Interfaces to Simplify Connections

Figure 4-9 shows the ATM router connected to the testbench, with the signals grouped into interfaces.

Figure 4-9 Testbench – router diagram with interfaces



4.9.4 ATM Interfaces

Here are the `Rx` and `Tx` interfaces with modports and clocking blocks.

Sample 4.43 Rx interface

```
// Rx interface with modports and clocking block
interface Rx_if (input logic clk);
  logic [7:0] data;
  logic soc, en, clav, rclk;

  clocking cb @(posedge clk);
    output data, soc, clav;  // Directions are relative
    input  en;               // to the testbench
  endclocking : cb

  modport DUT (output en, rclk,
               input  data, soc, clav);

  modport TB (clocking cb);
endinterface : Rx_if
```

Sample 4.44 Tx interface

```
// Tx interface with modports and clocking block
interface Tx_if (input logic clk);
  logic [7:0] data;
  logic soc, en, clav, tclk;

  clocking cb @(posedge clk);
      input  data, soc, en;
      output clav;
  endclocking : cb

  modport DUT (output data, soc, en, tclk,
               input  clk, clav);

  modport TB (clocking cb);
endinterface : Tx_if
```

4.9.5 ATM Router Model Using an Interface

Here are the ATM router model and testbench, which need to specify the `modport` in their port connection list. Note that you put the `modport` name after the interface name, `Rx_if`.

Sample 4.45 ATM router model with interface using modports

```
module atm_router(Rx_if.DUT Rx0, Rx1, Rx2, Rx3,
                  Tx_if.DUT Tx0, Tx1, Tx2, Tx3,
          input logic clk, rst);
  ...
endmodule
```

4.9.6 ATM Top Level Netlist with Interfaces

The top netlist has shrunk considerably, along with the chances of making a mistake.

Sample 4.46 Top-level netlist with interface

```
module top;
  bit clk, rst;
  always #5 clk = !clk;

  Rx_if Rx0 (clk), Rx1 (clk), Rx2 (clk), Rx3 (clk);
  Tx_if Tx0 (clk), Tx1 (clk), Tx2 (clk), Tx3 (clk);

  atm_router a1 (Rx0, Rx1, Rx2, Rx3,              // or just (.*)
                 Tx0, Tx1, Tx2, Tx3, clk, rst);

  test       t1 (Rx0, Rx1, Rx2, Rx3,              // or just (.*)
                 Tx0, Tx1, Tx2, Tx3, clk, rst);
endmodule : top
```

4.9.7 ATM Testbench with Interface

Sample 4.47 shows the part of the testbench that captures cells coming in from the TX port of the router. Note that the interface names are hard-coded, and so you have to duplicate the same code four times for the 4 × 4 ATM router. Chapter 10 shows how to simplify the code by using virtual interfaces.

Sample 4.47 Testbench using an interface with a clocking block

```
program test(Rx_if.TB Rx0, Rx1, Rx2, Rx3,
             Tx_if.TB Tx0, Tx1, Tx2, Tx3,
             input logic clk, output logic rst);

  bit [7:0] bytes[ATM_CELL_SIZE];

  initial begin
    // Reset the device
    rst <= 1;
    Rx0.cb.data <= 0;
    ...
    receive_cell0();
    ...
    end

  task receive_cell0();
    @(Tx0.cb);
    Tx0.cb.clav <= 1;          // Assert ready to receive
    wait (Tx0.cb.soc == 1);    // Wait for Start of Cell

    for (int i=0; i<ATM_CELL_SIZE; i++) begin
      wait (Tx0.cb.en == 0);  // Wait for enable
        @(Tx0.cb);

      bytes[i] = Tx0.cb.data;
      @(Tx0.cb);
      Tx0.cb.clav <= 0;        // Deassert flow control
    end
  endtask : receive_cell0

endprogram : test
```

4.10 The Ref Port Direction

SystemVerilog introduces a new port direction: `ref`. You should be familiar with the `input`, `output`, and `inout` directions. The last is for modeling bidirectional connections. If you drive a signal with multiple `inout` ports, SystemVerilog will calculate the value of the signal by combining the values of all drivers, taking in to effect driver strengths and Z values.

A `ref` port is a different beast. It is essentially a hierarchical reference to a variable (never a net) so that the value of the variable is the one last assigned. If you connect a variable to multiple `ref` ports, you may get race conditions as the port assignments from multiple modules update the single variable.

4.11 The End of Simulation

As described earlier in Section 4.4.6, simulation ends when the last `initial` block ends in the program. What really happens is that when the last `initial` block completes, it implicitly calls `$exit` to signify that this program is done. When every program has exited, an implicit call to `$finish` is done. Or, you can just call `$finish` anytime you want to end the simulation.

However, simulation is not yet over. A module or program can have one or more `final` blocks that contain code to be run just before the simulator terminates. This is a great place to perform clean up tasks such as closing files, and printing a report of the number of errors and warnings encountered. You cannot schedule any events, or have any delays in a `final` block. Note that you do not have to worry about freeing any memory that was allocated as this will be done automatically.

Sample 4.48 A `final` block

```
program test;
  int errors, warnings;

  initial begin
    ... // Main program activity
  end

  final
    $display("Test done with %0d errors and %0d warnings",
             errors, warnings);
endprogram
```

4.12 Directed Test for the LC3 Fetch Block

With what you have learned so far, you can create a simple test of a block of a larger design. The rest of this chapter shows a directed test for a block of the LC3 microcontoller. Later chapters show you how to create random tests, and tests structured using OOP.

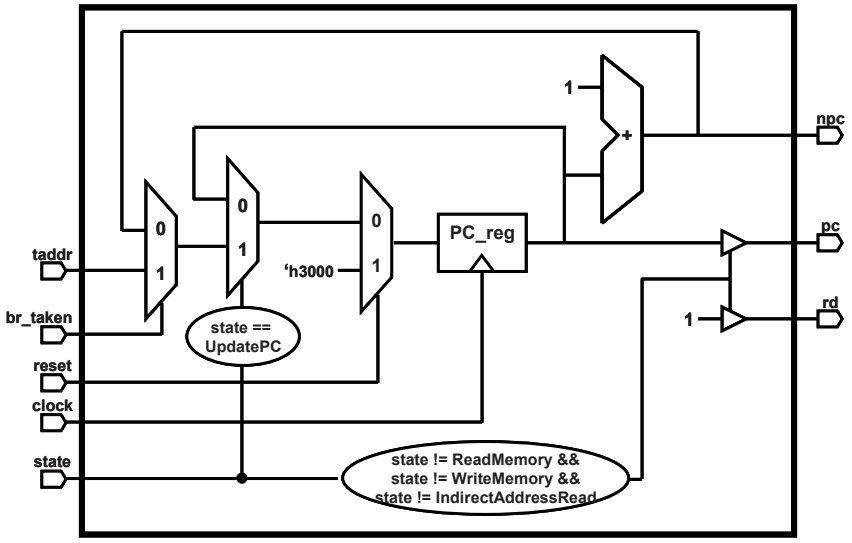
The Little Computer 3 (LC3) is an assembly language for teaching the fundamentals of programming to computer science and computer engineering students. The LC3 was developed by Yale N. Patt at the University of Texas at Austin and Sanjay J. Patel at the University of Illinois at Urbana-Champaign in the second edition of their textbook, Introduction to Computing Systems: From Bits and Gates to C and Beyond, Patt and Patel (2003).

Dr. Xun Liu, Dr. Rhett Davis, and Dr. Paul Franzon of North Carolina State University have implemented the LC3 for ECE 406, “Design of Complex Digital Systems.”

You can download the design specification and the protected Verilog code from `http://chris.spear.net/systemverilog`.

The LC3 implementation contains six blocks: `fetch`, `execute`, `writeback`, `memAccess`, `decode`, and `controller`. It implements the following instructions: ADD, AND, NOT, BR, JMP, LD, LDR, LDI, LSE, ST, STR, and STI.

Figure 4-10 LD3 Microcontroller fetch block



The `fetch` block in Figure 4-10 computes the addresses of instructions to fetch from memory. It has the following inputs:

- `clock`, `reset`: 1-bit
- `br_taken`: 1-bit. Tells the `fetch` block that a control signal has been encountered and thus `npc` is going to change from `pc`+1 to `taddr` (target address) as computed by the instruction
- `taddr`: 16-bits. The target instruction computed for a branch or jump instruction.
- `state`: 4-bits. The current state of the `controller` block, such as `fetch`, `decode`, etc.

The `fetch` block has the following outputs:

- `rd`: 1-bit. Tells the memory to perform a read. This signal is high impedance (Z) during the states `ReadMemory`, `WriteMemory`, and `IndirectAd`

dressRead as the memAccess block drives the shared bus during these cycles. During all other states, rd should be high.

- pc: 16-bits. The current value of the program counter register, PC_reg, or high impedance when rd is high impedance.
- npc: 16-bits. Always PC_reg+1.

At the positive edge of clock, when br_taken is true, PC_reg is updated with taddr, and with npc with br_taken is false. PC_reg is reset to 16'h3000. All signals are created on one clock cycle.

The Verilog code for the fetch block has input and output ports.

Sample 4.49 Fetch block Verilog code

```
module fetch(clock, reset, state, pc, npc, rd,
             taddr, br_taken);
  input clock, reset, br_taken;
  input [15:0] taddr;
  input [3:0] state;
  output [15:0] pc, npc;   // current and next PC
  output rd;

// protected code omitted
endmodule
```

The test uses an interface with clocking blocks to ensure that signals are read and sampled synchronously. There is a separate monitor modport so that the test can read back the values that have been written.

Sample 4.50 Fetch block interface

```
interface fetch_ifc(input bit clock);
   logic reset, br_taken, rd;
   logic [15:0] taddr;
   cntrl_e      state;             // Defined in Sample 4.52
   logic [15:0] pc, npc;           // current and next PC

   clocking cb @(posedge clock);
     input pc, npc, rd;
     output taddr, state, br_taken, reset;
   endclocking // cb

   modport TEST (clocking cb, output reset);

   modport DUT (
    input clock, reset, br_taken, taddr, state,
    output pc, npc, rd);

    // For monitoring DUT signals
   clocking cbm @(posedge clock);
     input pc, npc, rd, taddr, state, br_taken;
   endclocking // cbm
   modport MONITOR (clocking cbm);

endinterface // fetch_ifc
```

The directed test synchronously drives signals into the fetch block through the interface.

Sample 4.51 Fetch block directed test

```
program automatic test(fetch_ifc.TEST if_t,
                       fetch_ifc.MONITOR if_m);

  initial begin
    cntrl_e cntrl;

    $timeformat(-9,0,"ns",5);
    $monitor("%t: pc=%h npc=%h rd=%b state=%s",
             $realtime, if_m.cbm.pc, if_m.cbm.npc,
             if_m.cbm.rd, if_m.cbm.state.name);

    $display("%t: Reset all signals", $realtime);
    if_t.reset <= 1;
    if_t.cb.taddr <= 16'hFFFC;
    if_t.cb.br_taken <= 0;
    if_t.cb.state <= CNTRL_UPDATE_PC;

    repeat (2) @if_t.cb;
    pc_post_reset: assert (if_t.cb.pc == 16'h3000);

    ##1 if_t.cb.reset <= 0; // Synchronously deassert reset

    @(if_t.cb);
    $display("\n%t: Test loading of target address",
             $realtime);
    if_t.cb.state <= CNTRL_UPDATE_PC;
    if_t.cb.br_taken <= 1;

    @(if_t.cb);
    @(if_t.cb);
    pc_br_taken: assert (if_t.cb.pc == 16'hFFFC);

    $display("%t: Did the PC rollover as expected?",
             $realtime);
    if_t.cb.br_taken <= 0;
    if_t.cb.state <= CNTRL_UPDATE_PC;
    repeat (5) @(if_t.cb);
    pc_rollover: assert (if_t.cb.pc == 16'h0000);

    $display("\n%t: Step through all the controller states",
             $realtime);
    for (int i=CNTRL_FETCH; i<=CNTRL_COMPUTE_MEM; i++)
      begin
      $cast(cntrl, i);
      if (cntrl == CNTRL_UPDATE_PC)
        continue;
```

```
      $display("%t: Try with controller state=%0d %s",
               $realtime, cntrl, cntrl.name);
      if_t.cb.br_taken <= 0;
      if_t.cb.state <= cntrl;
      repeat (2) @(if_t.cb);
      pc_no_load: assert (if_t.cb.pc == 16'h0001);
    end // for i

    $display("\n%t: Tristate on PC output", $realtime);
    if_t.cb.state <= CNTRL_READ_MEM;
    @(if_t.cb);
    pc_z_read_mem: assert (if_t.cb.pc === 16'hzzzz);

    if_t.cb.state <= CNTRL_IND_ADDR_RD;
    @(if_t.cb);
    pc_z_ind_addr_rd: assert (if_t.cb.pc === 16'hzzzz);

    if_t.cb.state <= CNTRL_WRITE_MEM;
    @(if_t.cb);
    pc_z_write_mem: assert (if_t.cb.pc === 16'hzzzz);
   end
endprogram // test
```

The top level block instantiates the fetch interface, the `fetch` block and the test. It also defines the controller state enumerated type and so it can be used in both the test and interface.

Sample 4.52 Top level block for fetch testbench

```
`timescale 1ns/1ns

typedef enum {CNTRL_UPDATE_PC   = 0,
              CNTRL_FETCH       = 1,
              CNTRL_DECODE      = 2,
              CNTRL_EXECUTE     = 3,
              CNTRL_UPDATE_REGF = 4,
              CNTRL_COMPUTE_PC  = 5,
              CNTRL_COMPUTE_MEM = 6,
              CNTRL_READ_MEM    = 7,
              CNTRL_IND_ADDR_RD = 8,
              CNTRL_WRITE_MEM   = 9} cntrl_e;

module top;
   bit clock;
   always #10 clock = ~clock;

   fetch_ifc fif(clock);
   test  t1(fif, fif);
   fetch f1(clock, fif.reset, fif.state, fif.pc,
            fif.npc, fif.rd, fif.taddr, fif.br_taken);

endmodule // top
```

4.13 Conclusion

In this chapter you have learned how to use SystemVerilog's interfaces to organize the communication between design blocks and your testbench. With this design construct, you can replace dozens of signal connections with a single interface, making your code easier to maintain and improve, and reducing the number of wiring mistakes.

SystemVerilog also introduces the program block to hold your testbench and to reduce race conditions between the device under test and the testbench. With a clocking block in an interface, your testbenches will drive and sample design signals correctly relative to the clock.

Chapter 5

Basic OOP

5.1 Introduction

With procedural programming languages such as Verilog and C, there is a strong division between data structures and the code that uses them. The declarations and types of data are often in a different file than the algorithms that manipulate them. As a result, it can be difficult to understand the functionality of a program, as the two halves are separate.

Verilog users have it even worse than C users, as there are no structures in Verilog, only bit vectors and arrays. If you wanted to store information about a bus transaction, you would need multiple arrays: one for the address, one for the data, one for the command, and more. Information about transaction N is spread across all the arrays. Your code to create, transmit, and receive transactions is in a module that may or may not be actually connected to the bus. Worst of all, the arrays are all static, and so if your testbench only allocated 100 array entries, and the current test needed 101, you would have to edit the source code to change the size and recompile. As a result, the arrays are sized to hold the greatest conceivable number of transactions, but during a normal test, most of that memory is wasted.

Object-Oriented Programming (OOP) lets you create complex data types and tie them together with the routines that work with them. You can create testbenches and system-level models at a more abstract level by calling routines to perform an action rather than toggling bits. When you work with transactions instead of signal transitions, you are more productive. As a bonus, your testbench is decoupled from the

design details, making it more robust and easier to maintain and reuse on future projects.

If you already are familiar with OOP, skim this chapter, as SystemVerilog follows OOP guidelines fairly closely. Be sure to read Section 5.18 to learn how to build a testbench. Chapter 8 presents advanced OOP concepts such as inheritance and more testbench techniques; it should be read by everyone.

5.2 Think of Nouns, not Verbs

Grouping data and code together helps you in creating and maintaining large testbenches. How should data and code be brought together? You can start by thinking of how you would perform the testbench's job.

The goal of a testbench is to apply stimulus to a design and then check the result to see if it is correct. The data that flows into and out of the design is grouped together into transactions. The best way to organize the testbench is around the transactions, and the operations that you perform on them. In OOP, the transaction is the focus of your testbench.

You can think of an analogy between cars and testbenches. When you get into a car, you want to perform discrete actions, such as starting, moving forward, turning, stopping, and listening to music while you drive. Early cars required detailed knowledge about their internals to operate. You had to advance or retard the spark, open and close the choke, keep an eye on the engine speed, and be aware of the traction of the tires if you drove on a slippery surface such as a wet road. Today your interactions with the car are at a high level. If you want to start a car, just turn the key in the ignition, and you are done. Get the car moving by pressing the gas pedal; stop it with the brakes. Are you driving on snow? Don't worry: the anti-lock brakes help you stop safely and in a straight line.

Your testbench should be structured the same way. Traditional testbenches were oriented around the operations that had to happen: create a transaction, transmit it, receive it, check it, and make a report. Instead, you should think about the structure of the testbench, and what each part does. The generator creates transactions and passes them to the next level. The driver talks with the design that responds with transactions that are received by a monitor. The scoreboard checks these against the expected data. You should divide your testbench into blocks, and then define how they communicate.

5.3 Your First Class

A class encapsulates the data together with the routines that manipulate it. Sample 5.1 shows a class for a generic packet. The packet contains an address, a CRC, and an

array of data values. There are two routines in the `Transaction` class: a function to display the packet address, and another that computes the CRC (cyclic redundancy check) of the data.



To make it easier to match the beginning and end of a named block, you can put a label on the end of it. In Sample 5.1 these end labels may look redundant, but in real code with many nested blocks, the labels help you find the mate for a simple end or `endtask`, `end function`, or `endclass`.

Sample 5.1 Simple transaction class

```
class Transaction;
  bit [31:0] addr, crc, data[8];

  function void display;
    $display("Transaction: %h", addr);
  endfunction : display

  function void calc_crc;
    crc = addr ^ data.xor;
  endfunction : calc_crc

endclass : Transaction
```



Every company has its own naming style. This book uses the following convention: Class names start with a capital letter and avoid using underscores, as in `Transaction` or `Packet`. Constants are all upper case, as in `CELL_SIZE`, and variables are lower case, as in `count` or `trans_type`. You are free to use whatever style you want.

5.4 Where to Define a Class

You can define a class in SystemVerilog in a `program`, `module`, `package`, or outside of any of these. Classes can be used in programs and modules. This book only shows classes that are used in a program block, as introduced in Chap. 4. Until then, think of a program block as a module that holds your test code. The program holds a single test and contains the objects that comprise the testbench, and the initial blocks to create, initialize, and run the test.

When you start a project, you may want to store a single class per file. When the number of files gets too large, you can group a set of related classes and type definitions into a SystemVerilog `package`. For instance, you might group together all SCSI/ATA transactions into a single `package`. Now you can compile the package separately from the rest of the system. Unrelated classes, such as those for transactions, scoreboards, or different protocols, should remain in separate files.

See the SystemVerilog LRM for more information on packages.

5.5 OOP Terminology

What separates you, an OOP novice, from an expert? The first thing is the words you use. You already know some OOP concepts from working with Verilog. Here are some OOP terms, definitions, and rough equivalents in Verilog-2001.

- Class – a basic building block containing routines and variables. The analogue in Verilog is a module.
- Object – an instance of a class. In Verilog, you need to instantiate a module to use it.
- Handle – a pointer to an object. In Verilog, you use the name of an instance when you refer to signals and methods from outside the module. An OOP handle is like the address of the object, but is stored in a pointer that can only refer to one type.
- Property – a variable that holds data. In Verilog, this is a signal such as a register or wire.
- Method – the procedural code that manipulates variables, contained in tasks and functions. Verilog modules have tasks and functions plus initial and always blocks.
- Prototype – the header of a routine that shows the name, type, and argument list. The body of the routine contains the executable code.

This book uses the more traditional terms from Verilog of "variable" and "routine" rather than OOP's "property" and "method." If you are comfortable with the OOP terms, you can skim this chapter.

In Verilog you build complex designs by creating modules and instantiating them hierarchically. In OOP you create classes and instantiate them (creating objects) to create a similar hierarchy.

Here is an analogy to explain these OOP terms. Think of a class as the blueprint for a house. This plan describes the structure of the house, but you cannot live in a blueprint; you need to build the physical house. An object is the actual house. Just as one set of blueprints can be used to build a whole subdivision of houses, a single class can be used to build many objects. The house address is like a handle in that it uniquely identifies your house. Inside your house you have things such as lights (on or off), with switches to control them. A class has variables that hold values, and routines that control the values. A class for the house might have many lights. A single call to `turn_on_porch_light()` sets the light variable ON in a single house.

5.6 Creating New Objects

Both Verilog and OOP have the concept of instantiation, but there are some differences in the details. A Verilog module, such as a counter, is instantiated when you compile your design. A SystemVerilog class, such as a network packet, is instantiated at run-time when needed by the testbench. Verilog instances are static, as the hardware does not change during simulation; only signal values change. Stimulus objects are constantly being created and used to drive the DUT and check the results. Later, the objects may be freed so that their memory can be used by new ones.[1]

The analogy between OOP and Verilog has a few other exceptions. The top-level Verilog module is not usually explicitly instantiated. However, a SystemVerilog class must be instantiated before it can be used. Next, a Verilog instance name only refers to a single instance, whereas a SystemVerilog handle can refer to many objects, though only one at a time.

5.6.1 No News is Good News

In Sample 5.2, **`tr`** is a handle that points to an object of type `Transaction`. You can simplify this by just calling **`tr`** a `Transaction` handle.

Sample 5.2 Declaring and using a handle

```
Transaction tr;  // Declare a handle
tr = new();  // Allocate a Transaction object
```

When you declare the handle `tr`, it is initialized to the special value `null`. Next, you call the `new()` function to construct the `Transaction` object. `new` allocates space for the `Transaction`, initializes the variables to their default value (0 for 2-state variables and X for 4-state ones), and returns the address where the object is stored. For every class, SystemVerilog creates a default `new` to allocate and initialize an object. See Section 5.6.2 for more details on this function.

5.6.2 Custom Constructor

Sometimes OOP terminology can make a simple concept seem complex. What does "instantiation" mean? When you call `new` to instantiate an object, you are allocating a new block of memory to store the variables for that object. For example, the `Transaction` class has two 32-bit registers (`addr` and `crc`) and an array with eight values (data), for a total of 10 longwords, or 40 bytes. So when you call `new`, SystemVerilog allocates 40 bytes of storage. If you have used C, this step is similar to the `malloc`

[1]Back to the house analogy: the address is normally static, unless your house burns down, causing you to construct a new one. And garbage collection is never automatic.

function. (Note that SystemVerilog uses additional memory for 4-state variables and housekeeping information such as the object's type.)

The constructor does more than allocate memory; it also initializes the values. By default, variables are set to their default values – 0 for 2-state variables and X for 4-state. You can define your own `new()` function to set your own values. That is why the `new()` function is also called the "constructor," as it builds the object, just as your house is constructed from wood and nails. Note that you should not give a return value type as the constructor always returns a handle to an object of the same type as the class.

Sample 5.3 Simple user-defined new() function

```
class Transaction;
  logic [31:0] addr, crc, data[8];

  function new();
    addr = 3;
    foreach (data[i])
      data[i] = 5;
  endfunction

endclass
```

Sample 5.3 sets `addr` and `data` to fixed values but leaves `crc` at its default value of X. (SystemVerilog allocates the space for the object automatically.) You can use arguments with default values to make a more flexible constructor, as shown in Sample 5.4. Now you can specify the value for `addr` and `data` when you call the constructor, or use the default values.

Sample 5.4 A new() function with arguments

```
class Transaction;
  logic [31:0] addr, crc, data[8];

  function new(logic [31:0] a=3, d=5);
    addr = a;
    foreach (data[i])
      data[i] = d;
  endfunction
endclass

initial begin
  Transaction tr;
  tr = new(10);  // data uses default of 5
end
```

How does SystemVerilog know which `new()` function to call? It looks at the type of the handle on the left side of the assignment. In Sample 5.5, the call to `new` inside the

`Driver` constructor calls the `new()` function for `Transaction`, even though the one for `Driver` is closer. Since `tr` is a `Transaction` handle, SystemVerilog does the right thing and creates an object of type `Transaction`.

Sample 5.5 Calling the right new() function

```
class Transaction;
  ...
endclass : Transaction

class Driver;
  Transaction tr;
  function new();       // DriverÕs new function
    tr = new();         // Call the Transaction new function
  endfunction
endclass : Driver
```

5.6.3 Separating the Declaration and Construction



You should avoid declaring a handle and calling the constructor, `new`, all in one statement. While this is legal syntax and less verbose, it can create ordering problems, as the constructor is called before the first procedural statement. You might need to initialize objects in a certain order, but if you call `new`() in the declaration, you won't have the same control. Additionally, if you forget to use `automatic` storage, the constructor is called at the start of simulation, not when the block is entered.

5.6.4 The Difference Between New() and New[]

You may have noticed that this `new()` function looks a lot like the `new[]` operator, described in Section 2.3, used to set the size of dynamic arrays. They both allocate memory and initialize values. The big difference is that the `new()` function is called to construct a single object, whereas the `new[]` operator is building an array with multiple elements. `new()` can take arguments for setting object values, whereas `new[]` only takes a single value for the number of elements in the array.

5.6.5 Getting a Handle on Objects



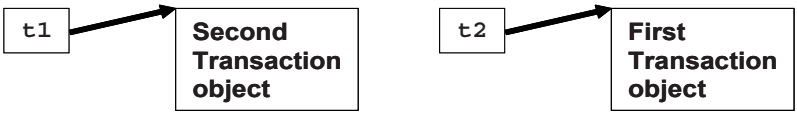
New OOP users often confuse an object with its handle. The two are very distinct. You *declare* a handle and *construct* an object. Over the course of a simulation, a handle can point to many objects. This is the dynamic nature of OOP and SystemVerilog. Don't get the handle confused with the object.

In Sample 5.6, `t1` first points to one object, then another. Figure 5-1 shows the resulting handles and objects.

Sample 5.6 Allocating multiple objects

```
Transaction t1, t2;   // Declare two handles
t1 = new();           // Allocate first Transaction object
t2 = t1;              // t1 & t2 point to it
t1 = new();           // Allocate second Transaction object
```

Figure 5-1 Handles and objects after allocating multiple objects



Why would you want to create objects dynamically? During a simulation you may need to create hundreds or thousands of transactions. SystemVerilog lets you create new ones automatically, when you need them. In Verilog, you would have to use a fixed-size array large enough to hold the maximum number of transactions.

Note that this dynamic creation of objects is different from anything else offered before in the Verilog language. An instance of a Verilog module and its name are bound together statically during compilation. Even with `automatic` variables, which come and go during simulation, the name and storage are always tied together.

An analogy for handles is people who are attending a conference. Each person is similar to an object. When you arrive, a badge is "constructed" by writing your name on it. This badge is a handle that can be used by the organizers to keep track of each person. When you take a seat for the lecture, space is allocated. You may have multiple badges for attendee, presenter, or organizer. When you leave the conference, your badge may be reused by writing a new name on it, just as a handle can point to different objects through assignment. Lastly, if you lose your badge and there is nothing to identify you, you will be asked to leave. The space you take, your seat, is reclaimed for use by someone else.

5.7 Object Deallocation

Now you know how to create an object – but how do you get rid of it? For example, your testbench creates and sends thousands of transactions such as transactions into your DUT. Once you know the transaction completed successfully, and you gather statistics, you don't need to keep it around. You should reclaim the memory; otherwise, a long simulation might run out of memory, or at least run more and more slowly.

Garbage collection is the process of automatically freeing objects that are no longer referenced. One way SystemVerilog can tell if an object is no longer being used is by

keeping track of the number of handles that point to it. When the last handle no longer references an object, SystemVerilog releases the memory for it.[2]

Sample 5.7 Creating multiple objects

```
Transaction t;  // Create a handle
t = new();      // Allocate a new Transaction
t = new();      // Allocate a second one, free the first
t = null;       // Deallocate the second
```

The second line in Sample 5.7 calls `new()` to construct an object and store the address in the handle `t`. The next call to `new()` constructs a second object and stores its address in `t`, overwriting the previous value. Since there are no handles pointing to the first object, SystemVerilog can deallocate it. The object may be deleted immediately, or wait a short wait. The last line explicitly clears the handle so that now the second object can be deallocated.

If you are familiar with C++, these concepts of objects and handles might look familiar, but there are some important differences. A SystemVerilog can handle only point to objects of one type, and so they are called "type-safe." In C, a typical void pointer is only an address in memory, and you can set it to any value or modify it with operators such as pre-increment. You cannot be sure that a pointer really is valid. A C++ typed pointer is much safer, but you may be tempted by C's flexibility. SystemVerilog does not allow any modification of a handle or using a handle of one type to refer to an object of another type. (SystemVerilog's OOP specification is closer to Java than C++.)

Secondly, since SystemVerilog performs automatic garbage collection when no more handles refer to an object, you can be sure your code always uses valid handles. In C / C++, a pointer can refer to an object that no longer exists. Garbage collection in those languages is manual, and so your code can suffer from "memory leaks" when you forget to deallocate objects.



SystemVerilog cannot garbage collect an object that is still referenced somewhere by a handle. For example, if you keep objects in a linked list, SystemVerilog cannot deallocate the objects until you manually clear all handles by setting them to `null`. If an object contains a routine that forks off a thread, the object is not deallocated while the thread is running. Likewise, any objects that are used by a spawned thread may not be deallocated until the thread terminates. See Chap. 7 for more information on threads.

[2]The actual algorithm to find unused objects varies between simulators. This section describes reference counting, which is the easiest to understand.

5.8 Using Objects

Now that you have allocated an object, how do you use it? Going back to the Verilog module analogy, you can refer to variables and routines in an object with the "." notation as shown in Sample 5.8.

Sample 5.8 Using variables and routines in an object

```
Transaction t;      // Declare a handle to a Transaction
t = new();          // Construct a Transaction object
t.addr = 32Õh42;    // Set the value of a variable
t.display();        // Call a routine
```

In strict OOP, the only access to variables in an object should be through its public methods such as `get()` and `put()`. This is because accessing variables directly limits your ability to change the underlying implementation in the future. If a better (or simply different) algorithm comes along in the future, you may not be able to adopt it because you would also need to modify all of the references to the variables.



The problem with this methodology is that it was written for large software applications with lifetimes of a decade or more. With dozens of programmers making modifications, stability is paramount. However, you are creating a testbench, where the goal is maximum control of all variables to generate the widest range of stimulus values. One of the ways to accomplish this is with constrained-random stimulus generation, which cannot be done if a variable is hidden behind a screen of methods. While the `get()` and `put()` methods are fine for compilers, GUIs, and APIs, you should stick with public variables that can be directly accessed anywhere in your testbench.

5.9 Static Variables vs. Global Variables

Every object has its own local variables that are not shared with any other object. If you have two `Transaction` objects, each has its own `addr`, `crc`, and `data` variables. Sometimes though, you need a variable that is shared by all objects of a certain type. For example, you might want to keep a running count of the number of transactions that have been created. Without OOP, you would probably create a global variable. Then you would have a global variable that is used by one small piece of code, but is visible to the entire testbench. This "pollutes" the global name space and makes variables visible to everyone, even if you want to keep them local.

5.9.1 A Simple Static Variable

In SystemVerilog you can create a static variable inside a class. This variable is shared amongst all instances of the class, but its scope is limited to the class. In Sample 5.9, the static variable `count` holds the number of objects created so far. It is initialized to 0 in the declaration because there are no transactions at the beginning of the simulation. Each time a new object is constructed, it is tagged with a unique value, and `count` is incremented.

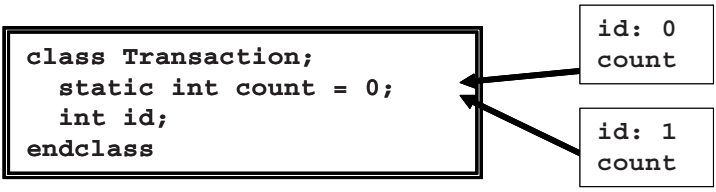
Sample 5.9 Class with a static variable

```
class Transaction;
  static int count = 0; // Number of objects created
  int id;               // Unique instance ID
  function new();
    id = count++;       // Set ID, bump count
  endfunction
endclass

Transaction t1, t2;
initial begin
  t1 = new();           // 1st instance, id=0, count=1
  t2 = new();           // 2nd instance, id=1, count=2
  $display("Second id=%d, count=%d", t2.id, t2.count);
end
```

In Sample 5.9, there is only one copy of the static variable `count`, regardless of how many `Transaction` objects are created. You can think that `count` is stored with the class and not the object. The variable `id` is not static, and so every `Transaction` has its own copy, as shown in Figure 5-2. Now you don't need to make a global variable for the count.

Figure 5-2 Static variables in a class





Using the ID field is a good way to track objects as they flow through a design. When debugging a testbench, you often need a unique value. SystemVerilog does not let you print the address of an object, but you can make an ID field. Whenever you are tempted to make a global variable, consider making a class-level static variable. A class should be self-contained, with as few outside references as possible.

5.9.2 Accessing Static Variables Through the Class Name

Sample 5.9 showed how you can reference a static variable using a handle. You don't need a handle; you could use the class name followed by `::`, the class scope resolution operator as shown in Sample 5.10

Sample 5.10 The class scope resolution operator

```
class Transaction;
  static int count = 0;         // Number of objects created
  ...
endclass

initial begin
  run_test();
  $display("%d transaction were created",
           Transaction::count);  // Reference static w/o handle
end
```

5.9.3 Initializing Static Variables

A static variable is usually initialized in the declaration. You can't easily initialize it in the class constructor, as this is called for every single new object. You would need another static variable to act as a flag, indicating whether the original variable had been initialized. If you have a more elaborate initialization, you could use an initial block. Just make sure the static variables are initialized before the first object is constructed.

5.9.4 Static Methods

Another use for a static variable is when every instance of a class needs information from a single object. For example, a transaction class may refer to a configuration object for a mode bit. If you have a nonstatic handle in the `Transaction` class, every object will have its own copy, wasting space. Sample 5.11 shows how to use a static variable instead.

Sample 5.11 Static storage for a handle

```
class Transaction;
  static Config cfg;      // A handle with static storage
  MODE_E mode;

  function new();
    mode = cfg.mode;
  endfunction
endclass

Config cfg;
initial begin
  cfg = new(MODE_ON);
  Transaction::cfg = cfg;
  ...
end
```

As you employ more static variables, the code to manipulate them may grow into a full fledged routine. In SystemVerilog, you can create a static method inside a class that can read and write static variables, even before the first instance has been created.

Sample 5.12 has a simple static function to display the values of the static variables. SystemVerilog does not allow a static method to read or write nonstatic variables, such as `id`. You can understand this restriction based on the code below. When the function `display_statics` is called at the end of the example, no `Transaction` objects have been constructed, and so no storage has been created for `id` variables.

Sample 5.12 Static method displays static variable

```
class Transaction;
  static Config cfg;
  static int count = 0;
  int id;

  // Static method to display static variables.
  static function void display_statics();
    $display(ÒTransaction cfg.mode=%s, count=%0dÓ,
             cfg.mode.name(), count);
  endfunction
endclass

Config cfg;
initial begin
  cfg = new(MODE_ON);
  Transaction::cfg = cfg;
  Transaction::display_statics();  // Static method call
  end
```

5.10 Class Methods

A method in a class is just a `task` or `function` defined inside the scope of the class. Sample 5.13 defines `display()` methods for the `Transaction` and `PCI_Tran`. SystemVerilog calls the correct one, based on the handle type.

Sample 5.13 Routines in the class

```
class Transaction;
  bit [31:0] addr, crc, data[8];
  function void display();
    $display("@%0t: TR addr=%h, crc=%h", $time, addr, crc);
    $write("\tdata[0-7]=");
    foreach (data[i]) $write(data[i]);
    $display();
  endfunction
endclass

class PCI_Tran;
  bit [31:0] addr, data;  // Use realistic names
  function void display();
    $display("@%0t: PCI: addr=%h, data=%h", $time, addr, data);
  endfunction
endclass

Transaction t;
PCI_Tran pc;

initial begin
  t = new();      // Construct a Transaction
  t.display();    // Display a Transaction
  pc = new();     // Construct a PCI transaction
  pc.display();   // Display a PCI Transaction
end
```

A method in a class uses automatic storage by default, and so you don't have to worry about remembering the `automatic` modifier.

5.11 Defining Methods Outside of the Class



A good rule of thumb is you should limit a piece of code to one "page" to keep it understandable. You may be familiar with this rule for routines, but it also applies to classes. If you can see everything in a class on the screen at one time, you can more easily understand it.

However, if each method takes a page, how can the whole class fit on a page? In SystemVerilog you can break a method into the prototype (method name and arguments) inside the class, and the body (the procedural code) that goes after the class.

Here is how you create out-of-block declarations. Copy the first line of the method, with the name and arguments, and add the `extern` keyword at the beginning. Now take the entire method and move it after the class body, and add the class name and

two colons (: : the scope operator) before the method name. The above classes could be defined as follows.

Sample 5.14 Out-of-block method declarations

```
class Transaction;
  bit [31:0] addr, crc, data[8];
  extern function void display();
endclass

function void Transaction::display();
  $display("@%0t: Transaction addr=%h, crc=%h",
           $time, addr, crc);
  $write("\tdata[0-7]=");
  foreach (data[i]) $write(data[i]);
  $display();
endfunction

class PCI_Tran;
  bit [31:0] addr, data;  // Use realistic names
  extern function void display();
endclass

function void PCI_Tran::display();
  $display("@%0t: PCI: addr=%h, data=%h",
           $time, addr, data);
endfunction
```



A common coding mistake is when the method prototype does not match the one in the body. SystemVerilog requires that the prototype be identical to the out-of-block method declaration, except for the class name and scope operator. Additionally, some OOP compilers (g++ and VCS) prohibit you from specifying the default argument values in both the prototype and the body. Since default argument values are important to code that calls a method, not to its implementation, they should only be present in class declaration.



Another common mistake is to leave out the class name when you declare the method outside of the class. As a result, it is defined at the next higher scope (probably the program or package scope), and the compiler gives an error when the task tries to access class-level variables and methods. This is shown in Sample 5.15.

Sample 5.15 Out-of-body task missing class name

```
class Broken;
  int id;
  extern function void display;
endclass

function void display; // Missing Broken::
  $display("Broken: id=%0d", id); // Error, id not found
endfunction
```

5.12 Scoping Rules

When writing your testbench, you need to create and refer to many variables. SystemVerilog follows the same basic rules as Verilog, with a few helpful improvements.

A scope is a block of code such as a module, program, task, function, class, or `begin-end` block. The `for` and `foreach`-loops automatically create a block so that an index variable can be declared or created local to the scope of the loop.

You can define new variables in a block. New in SystemVerilog is the ability to declare a variable in an unnamed begin–end block, as shown in the `for`-loops that declare the index variable.

A name can be relative to the current scope or absolute starting with `$root`. For a relative name, SystemVerilog looks up the list of scopes until it finds a match. If you want to be unambiguous, use `$root` at the start of a name.[3]

Sample 5.16 uses the same name in several scopes. Note that in real code, you would use more meaningful names! The name `limit` is used for a global variable, a program variable, a class variable, a task variable, and a local variable in an initial block. The latter is in an unnamed block, and so the label created is tool dependent.

[3]These examples were tested with VCS 2006.06. The MTI Questa simulator uses $unit instead of $root. See Section 4.6 for more information on these two constructs.

Sample 5.16 Name scope

```
int limit;                    // $root.limit

program automatic p;
  int limit;                  // $root.p.limit

  class Foo;
    int limit, array[];       // $root.p.Foo.limit

    // $root.p.Foo.print.limit
    function void print (int limit);
      for (int i=0; i<limit; i++)
        $display("%m: array[%0d]=%0d", i, array[i]);
    endfunction
  endclass

  initial begin
    int limit = $root.limit; // **see note above
    Foo bar;

    bar = new();
    bar.array = new[limit];
    bar.print (limit);
    end
endprogram
```

For testbenches, you can declare variables in the `program` or in the `initial` block. If a variable is only used inside a single `initial` block, such as a counter, you should declare it there to avoid possible name conflicts with other blocks. Note that if you declare a variable in an unnamed block, such as the `initial` in Sample 5.16, there is no hierarchical name that works consistently across all tools.



Declare your classes outside of any `program` or `module` in a `package`. This approach can be shared by all the testbenches, and you can declare temporary variables at the innermost possible level. This style also eliminates a common bug that happens when you forget to declare a variable inside a class. SystemVerilog looks for that variable in higher scopes.



If a block uses an undeclared variable, and another variable with that name happens to be declared in the program block, the class uses it instead, with no warning. In Sample 5.17, the function `Bad::display` did not declare the loop variable `i`, and so SystemVerilog uses the program level `i` instead. Calling the function changes the value of `test.i`, probably not what you want!

Sample 5.17 Class uses wrong variable

```
program test;
  int i;  // Program-level variable

  class Bad;
    logic [31:0] data[];

    // Calling this function changes the program variable
    function void display;
      // Forgot to declare i in next statement
      for (i=0; i<data.size(); i++)
        $display("data[%0d]=%x", i, data[i]);
    endfunction
  endclass
endprogram
```

If you move the class into a package, the class cannot see the program-level variables, and thus won't use them unintentionally.

Sample 5.18 Move class into package to find bug

```
package Mistake;
  class Bad;
    logic [31:0] data[];

    // Will not compile because of undeclared i
    function void display;
      for (i = 0; i<data.size(); i++)
        $display("data[%0d]=%x", i, data[i]);
    endfunction
  endclass
endpackage

program test;
  int i;  // Program-level variable
  import Mistake::*;
  ...
endprogram
```

5.12.1 What is `this`?

When you use a variable name, SystemVerilog looks in the current scope for it, and then in the parent scopes until the variable is found. This is the same algorithm used by Verilog. What if you are deep inside a class and want to unambiguously refer to a class-level object? This style code is most commonly used in constructors, where the programmer uses the same name for a class variable and an argument.[4] In Sample 5.19, the

[4]Some people think this makes the code easier to read; others think it is a shortcut by a lazy programmer.

keyword "this" removes the ambiguity to let SystemVerilog know that you are assigning the local variable, oname, to the class variable, oname.

Sample 5.19 Using this to refer to class variable

```
class Scoping;
  string oname;

  function new(string oname);
    this.oname = oname;    // class oname = local oname
  endfunction

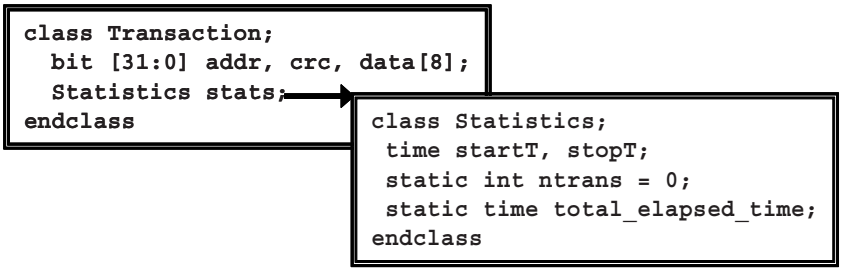
endclass
```

5.13 Using One Class Inside Another

A class can contain an instance of another class, using a handle to an object. This is just like Verilog's concept of instantiating a module inside another module to build up the design hierarchy. Common reasons for using containment are reuse and controlling complexity.

For example, every one of your transactions may have a statistics block, with timestamps on when the transaction started and ended, and information about all transactions, as shown in Figure 5-3.

Figure 5-3 Contained objects



Sample 5.20 shows the Statistics class.

Sample 5.20 `Statistics` class declaration

```
class Statistics;
  time startT, stopT;       // Transaction times
  static int ntrans = 0;    // Transaction count
  static time total_elapsed_time = 0;

  function time how_long;
    how_long = stopT Ð startT;
    ntrans++;
    total_elapsed_time += how_long;
  endfunction

  function void start;
    startT = $time;
  endfunction
endclass
```

Now you can use this class inside another.

Sample 5.21 Encapsulating the `Statistics` class

```
class Transaction;
  bit [31:0] addr, crc, data[8];
  Statistics stats;         // Statistics handle

  function new();
    stats = new();          // Make instance of stats
  endfunction

  task create_packet();
    // Fill packet with data
    stats.start();
    // Transmit packet
  endtask
endclass
```

The outermost class, `Transaction`, can refer to things in the `Statistics` class using the usual hierarchical syntax, such as `stats.startT`.

Remember to instantiate the object; otherwise, the handle `stats` is `null` and the call to `start` fails. This is best done in the constructor of the outer class, `Transaction`.

As your classes become larger, they may become hard to manage. When your variable declarations and method prototypes grow larger than a page, you should see if there is a logical grouping of items in the class so that it can be split into several smaller ones.

This is also a potential sign that it's time to refactor your code, i.e., split it into several smaller, related classes. See Chap. 8 for more details on class inheritance. Look at

what you're trying to do in the class. Is there something you could move into one or more base classes, i.e., decompose a single class into a class hierarchy? A classic indication is similar code appearing at various places in the class. You need to factor that code out into a function in the current class, one of the current class's parent classes, or both.

5.13.1 How Big or Small Should My Class Be?



Just as you may want to split up classes that are too big, you should also have a lower limit on how small a class should be. A class with just one or two members makes the code harder to understand as it adds an extra layer of hierarchy and forces you to constantly jump back and forth between the parent class and all the children to understand what it does. In addition, look at how often it is used. If a small class is only instantiated once, you might want to merge it into the parent class.

One Synopsys customer put each transaction variable into its own class for fine control of randomization. The transaction had a separate object for the address, CRC, data, etc. In the end, this approach only made the class hierarchy more complex. On the next project they flattened the hierarchy.

See Section 8.4 for more ideas on partitioning classes.

5.13.2 Compilation Order Issue

Sometimes you need to compile a class that includes another class that is not yet defined. The declaration of the handle causes an error, as the compiler does not recognize the new type. Declare the class name with a `typedef` statement, as shown below.

Sample 5.22 Using a `typedef class` statement

```
typedef class Statistics;  // Define a lower level class

class Transaction;
  Statistics stats;        // Use Statistics class
  ...
endclass

class Statistics;          // Define Statistics class
  ...
endclass
```

5.14 Understanding Dynamic Objects

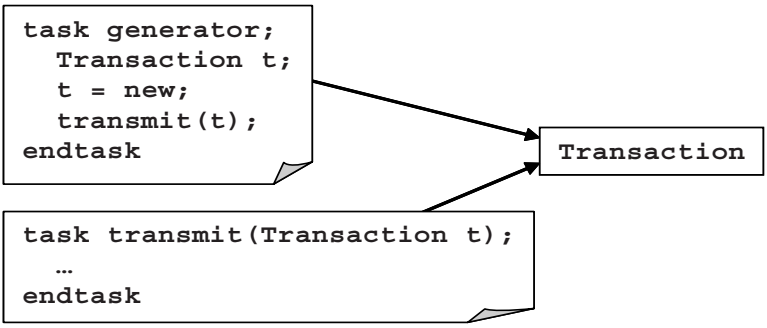
In a statically allocated language such as Verilog, every piece of data usually has a variable associated with it. For example, there may be a wire called `grant`, the integer `count`, and a module instance `i1`. In OOP, there is not the same one-to-one correspondence. There can be many objects, but only a few named handles. A testbench may allocate a thousand transaction objects during a simulation, but may only have a few handles to manipulate them. This situation takes some getting used to if you have only written Verilog code.

In reality, there is a handle pointing to every object. Some handles may be stored in arrays or queues, or in another object, like a linked list. For objects stored in a mailbox, the handle is in an internal SystemVerilog structure. See Section 7.6 for more information on mailboxes.

5.14.1 Passing Objects to Methods

What happens when you pass an object into a method? Perhaps the method only needs to read the values in the object, such as `transmit` above. Or, your method may modify the object, like a method to create a packet. Either way, when you call the method, you pass a handle to the object, not the object itself.

Figure 5-4 Handles and objects across methods



In Figure 5-4, the `generator` task has just called `transmit`. There are two handles, `generator.t` and `transmit.t`, that both refer to the same object.

When you call a method with a scalar variable (not an array or object) and use the `ref` argument keyword, SystemVerilog passes the address of the scalar, and so the method can modify it. If you don't use `ref`, SystemVerilog copies the scalar's value into the argument variable, and so any changes to the argument don't affect the original value.

Sample 5.23 Passing objects

```
// Transmit a packet onto a 32-bit bus
task transmit(Transaction t);
  CBbus.rx_data <= t.data;
  t.stats.startT = $time;
  ...
endtask

Transaction t;
initial begin
  t = new();        // Allocate the object
  t.addr = 42;      // Initialize values
  transmit(t);      // Pass object to task
end
```

In Sample 5.23, the initial block allocates a `Transaction` object and calls the `transmit` task with the handle that points to the object. Using this handle, `transmit` can read and write values in the object. However, if `transmit` tries to modify the handle, the result won't be seen in the initial block, as the `t` argument was not declared as `ref`.



A method can modify an object, even if the handle argument does not have a `ref` modifier. This frequently causes confusion for new users, as they mix up the handle with the object. As shown above, `transmit` can write a timestamp into the object without changing the value of `t`. If you don't want an object modified in a method, pass a copy of it so that the original object is untouched. See Section 5.15 for more on copying objects.

5.14.2 Modifying a Handle in a Task



A common coding mistake is to forget to use `ref` on method arguments that you want to modify, especially handles. In Sample 5.24, the argument `tr` is not declared as `ref`, and so any change to it is not be seen by the calling code. The argument `tr` has the default direction of `input`.

Sample 5.24 Bad transaction creator task, missing `ref` on handle

```
function void create(Transaction tr); // Bug, missing ref
  tr = new();
  tr.addr = 42;
  // Initialize other fields
  ...
endfunction

Transaction t;
initial begin
  create(t);            // Create a transaction
  $display(t.addr);     // Fails because t=null
  end
```

Even though `create` modified the argument `tr`, the handle `t` in the calling block remains `null`. You need to declare the argument `tr` as `ref`.

Sample 5.25 Good transaction creator task with `ref` on handle

```
function void create(ref Transaction tr);
  ...
endtask
```

5.14.3 Modifying Objects in Flight



A very common mistake is forgetting to create a new object for each transaction in the testbench. In Sample 5.26, the `generate_bad` task creates a `Transaction` object with random values, and transmits it into the design over several cycles.

Sample 5.26 Bad generator creates only one object

```
task generator_bad(int n);
  Transaction t;
  t = new();                    // Create one new object
  repeat (n) begin
    t.addr = $random();         // Initialize variables
    $display("Sending addr=%h", t.addr);
    transmit(t);                // Send it into the DUT
  end
endtask
```

What are the symptoms of this mistake? The code above creates only one `Transaction`, and so every time through the loop, `generator_bad` changes the object at the same time it is being transmitted. When you run this, the `$display` shows many `addr` values, but all transmitted `Transaction` objects have the same value of `addr`. The bug occurs when `transmit` spawns off a thread that takes several cycles to send

the transaction, and so the values in the object are re-randomized in the middle of transmission. If your `transmit` task makes a copy of the object, you can recycle the same object over and over. This bug can also happen with mailboxes as show in Sample 7.31.

To avoid this bug, you need to create a new `Transaction` during each pass through the loop.

Sample 5.27 Good generator creates many objects

```
task generator_good(int n);
  Transaction t;
  repeat (n) begin
    t = new();                  // Create one new object
    t.addr = $random();         // Initialize variables
    $display("Sending addr=%h", t.addr);
    transmit(t);                // Send it into the DUT
  end
endtask
```

5.14.4 Arrays of Handles

As you write testbenches, you need to be able to store and reference many objects. You can make arrays of handles, each of which refers to an object. Sample 5.28 shows storing ten bus transactions in an array.

Sample 5.28 Using an array of handles

```
task generator();
  Transaction tarray[10];
  foreach (tarray[i])
    begin
    tarray[i] = new();      // Construct each object
    transmit(tarray[i]);
    end
endtask
```

The array `tarray` is made of handles, not objects. So you need to construct each object in the array before using it, just as you would for a normal handle. There is no way to call **`new`** on an entire array of handles.

There is no such thing as an “array of objects,” though you may use this term as a shorthand for an array of handles that points to objects. You should keep in mind that some of these handles may not point to an object, or that multiple handles could point to a single object.

5.15 Copying Objects

You may want to make a copy of an object to keep a method from modifying the original, or in a generator to preserve the constraints. You can either use the simple, built-in copy available with `new` operator or you can write your own for more complex classes. See Section 8.2 for more reasons why you should make a copy method.

5.15.1 Copying an Object with the `new` Operator

Copying an object with the `new` operator is easy and reliable. Memory for the new object is allocated and all variables from the existing object are copied. However any `new()` function that you may have defined is not called.

Sample 5.29 Copying a simple class with new

```
class Transaction;
  bit [31:0] addr, crc, data[8];
endclass

Transaction src, dst;
initial begin
  src = new();         // Create first object
  dst = new src;       // Make a copy with new operator
  end
```

This is a shallow copy, similar to a photocopy of the original, blindly transcribing values from source to destination. If the class contains a handle to another class, only the handle's value is copied by the `new` operator, not the lower level one. In Sample 5.30, the **`Transaction`** class contains a handle to the **`Statistics`** class, originally shown in Sample 5.20.

Sample 5.30 Copying a complex class with `new` operator

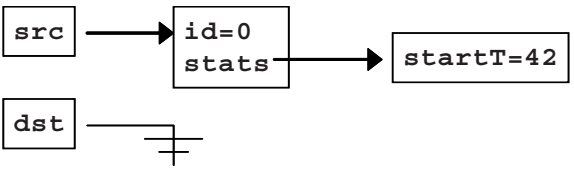
```
class Transaction;
  bit [31:0] addr, crc, data[8];
  static int count = 0;
  int id;
  Statistics stats;        // Handle points to Statistics object

  function new();
    stats = new();         // Construct a new Statistics obecjt
    id = count++;
  endfunction
endclass

Transaction src, dst;
initial begin
  src = new();            // Create a Transaction object
  src.stats.startT = 42; // Results in Figure 5-5
  dst = new src;          // Copy src to dst with new operator
                          // Results in Figure 5-6
  dst.stats.startT = 96; // Changes stats for dst & src
  $display(src.stats.startT); // Ò96Ó, see Figure 5-7
  end
```

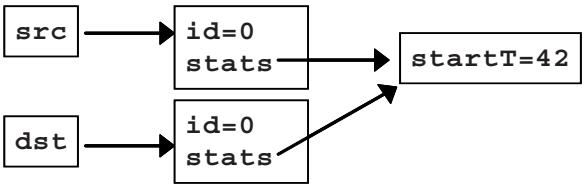
The initial block creates the first **`Transaction`** object and modifies a variable in the contained object **`Statistics`**., as shown in Figure 5-5.

Figure 5-5 Objects and handles before copy with the `new` operator



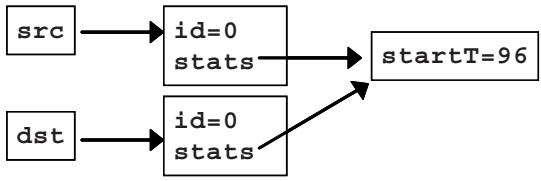
When you use the `new` operator to make a copy, the **`Transaction`** object is copied, but not the **`Statistics`** one. This is because the `new` operator does not call your own `new()` function. Instead, the values of variables and handles are copied. So now both **`Transaction`** objects have the same `id` as shown in Figure 5-6.

Figure 5-6 Objects and handles after copy with the `new` operator



Worse yet, both **Transaction** objects point to the same **Statistics** object and so modifying `startT` with the `src` handle affects what is seen with the `dst` handle.

Figure 5-7 Objects and handles after copy with the new operator



5.15.2 Writing Your Own Simple Copy Function

If you have a simple class that does not contain any references to other classes, writing a `copy` function is easy.

Sample 5.31 Simple class with copy function

```
class Transaction;
  bit [31:0] addr, crc, data[8];  // No Statistic handle

  function Transaction copy();
    copy = new();                 // Construct destination
    copy.addr = addr;             // Fill in data values
    copy.crc  = crc;
    copy.data = data;             // Array copy
  endfunction
endclass
```

Sample 5.32 Using a copy function

```
Transaction src, dst;
initial begin
  src = new();                    // Create first object
  dst = src.copy();               // Make a copy of the object
end
```

5.15.3 Writing a Deep Copy Function

For nontrivial classes, you should always create your own `copy` function. You can make it a deep copy by calling the `copy` functions of all the contained objects. Your own `copy` function makes sure all your user fields (such as an ID) remain consistent. The downside of making your own `copy` function is that you need to keep it up to

date as you add new variables – forget one and you could spend hours debugging to find the missing value.[5]

Sample 5.33 Complex class with deep copy function

```
class Transaction;
  bit [31:0] addr, crc, data[8];
  Statistics stats;        // Handle points to Statistics object
  static int count = 0;
  int id;

  function new();
    stats = new();
    id = count++;
  endfunction

  function Transaction copy();
    copy = new();              // Construct destination object
    copy.addr = addr;          // Fill in data values
    copy.crc  = crc;
    copy.data = data;
    copy.stats = stats.copy(); // Call Statistics::copy
  endfunction
endclass
```

The `new()` constructor is called by `copy` and so every object gets a unique `id`. Add a `copy()` method for the `Statistics` class, and every other class in the hierarchy.

Sample 5.34 `Statistics` class declaration

```
class Statistics;
  time startT, stopT;      // Transaction times
  ...                      // See Sample 5.20 for rest of class
  function Statistics copy();
    copy = new();
    copy.startT = startT
    copy.stopT = stopT;
  endfunction
endclass
```

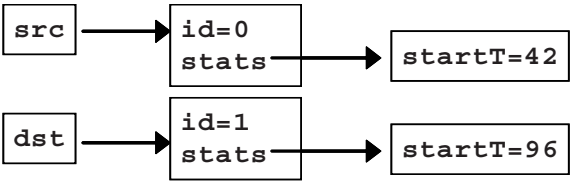
Now when you make a copy of the `Transaction` object, it will have its own `Statistics` object as shown in Sample 5.35.

[5]Perhaps the version of SystemVerilog may include a deep object copy. However, this still does just a copy, and so your constructor (`new` function) won't be called, and fields such as ID will not be updated.

Sample 5.35 Copying a complex class with `new operator`

```
Transaction src, dst;
initial begin
  src = new();            // Create first object
  src.stats.startT = 42; // Set start time
  dst = new src;          // Copy src to dst with deep copy
  dst.stats.startT = 96; // Changes stats for dst only
  $display(src.stats.startT); // Ò42Ó, See Figure 5-8
  end
```

Figure 5-8 Objects and handles after deep copy



5.15.4 Packing Objects to and from Arrays Using Streaming Operators

Some protocols, such as ATM, transmit control and data values one byte at a time. Before you send out a transaction, you need to pack together the variables in the object to a byte array. Likewise, after receiving a string of bytes, you need to unpack them back into a transaction object. For both of these functions, use the streaming operators, as originally shown in Section 2.11.3.

You can't just stream the entire object as this would gather all properties, including both data and also meta-data such as timestamps and self-checking information that you may not want packed. You need to write your own `pack` function that only uses the properties that you choose.

Sample 5.36 Transaction class with pack and unpack functions

```
class Transaction;
  bit [31:0] addr, crc, data[8]; // Real data
  static int count = 0;          // Meta-data does not
  int id;                        //     get packed

  function new();
    id = count++;
  endfunction

  function void display();
    $write("Tr: id=%0d, addr=%x, crc=%x", id, addr, crc);
    foreach(data[i]) $write(" %x", data[i]);
    $display;
  endfunction

  function void pack(ref byte bytes[40]);
    bytes = { >> {addr, crc, data}};
  endfunction

  function Transaction unpack(ref byte bytes[40]);
    { >> {addr, crc, data}} = bytes;
  endfunction
endclass : Transaction
```

Sample 5.37 Using the pack and unpack functions

```
Transaction tr, tr2;
byte b[40];                 // addr + crc + data = 40 bytes

initial begin
  tr = new();
  tr.addr = 32'ha0a0a0a0;   // Fill object with values
  tr.crc =  '1;
  foreach (tr.data[i])
    tr.data[i] = i;

  tr.pack(b);               // Pack object into byte array
  $write("Pack results: ");
  foreach (b[i])
    $write("%h", b[i]);
   $display;

  tr2 = new();
  tr2.unpack(b);
  tr2.display();
end
```

5.16 Public vs. Local

The core concept of OOP is encapsulating data and related methods into a class. Variables are kept local to the class by default to keep one class from poking around inside another. A class provides a set of accessor methods to access and modify the data. This would also allow you to change the implementation without needing to let the users of the class know. For instance, a graphics package could change its internal representation from Cartesian coordinates to polar as long as the user interface (accessor methods) have the same functionality.

Consider the `Transaction` class that has a payload and a CRC so that the hardware can detect errors. In conventional OOP, you would make a method to set the payload and also set the CRC so that they would stay synchronized. Thus your objects would always be filled with correct values.

However, testbenches are not like other programs, such as a web browser or word processor. A testbench needs to create errors. You want to have a bad CRC so that you can test how the hardware reacts to errors.

OOP languages such as C++ and Java allow you to specify the visibility of variables and methods. By default, everything in a class is local unless labeled otherwise.



In SystemVerilog, everything is public unless labeled `local` or `protected`. You should stick with this default so that you have the greatest control over the operation of the DUT, which is more important than long-term software stability. For example, making the CRC visible allows you to easily inject errors into the DUT. If the CRC was local, you would have to write extra code to bypass the data-hiding mechanisms, resulting in a larger and more complex testbench.

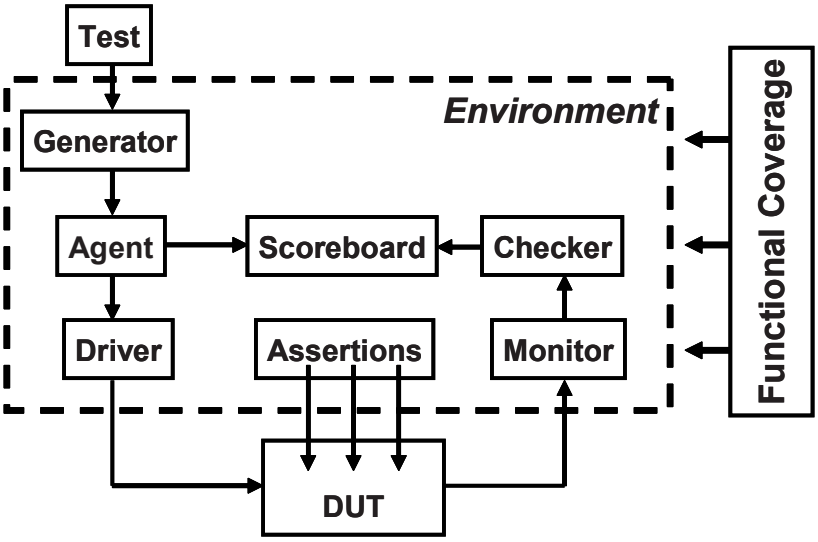
5.17 Straying Off Course

As a new OOP student, you may be tempted to skip the extra thought needed to group items into a class, and just store data in a few variables. Avoid the temptation! A basic DUT monitor samples several values from an interface. Don't just store them in some integers and pass them to the next stage. This saves you a few minutes at first, but eventually you need to group these values together to form a complete transaction. Several of these transactions may need to be grouped to create a higher-level transaction such as a DMA transfer. Instead, immediately put those interface values into a transaction class. Now you can store related information (port number, receive time) along with the data, and easily pass this object to the rest of your testbench.

5.18 Building a Testbench

You are closer to creating a simple testbench from classes. Here is the diagram from Chap. 1. Obviously, the transactions in Figure 5-9 are objects, but each block is represented as a class also.

Figure 5-9 Layered testbench



The `Generator`, `Agent`, `Driver`, `Monitor`, `Checker`, and `Scoreboard` are all classes, modeled as transactors (described below). They are instantiated inside the Environment class. For simplicity, the test is at the top of the hierarchy, as is the program that instantiates the Environment class. The Functional coverage definitions can be put inside or outside the Environment class.

A transactor is made of a simple loop that receives a transaction object from a previous block, makes some transformations, and sends it to the following one. Some, such as the `Generator`, have no upstream block, and so this transactor constructs and randomizes every transaction, while others, such as the `Driver`, receive a transaction and send it into the DUT as signal transitions.

Sample 5.38 Basic Transactor

```
class Transactor;  // Generic class
  Transaction tr;

  task run();
    forever begin
      // Get transaction from upstream block
      ...
      // Do some processing
      ...
      // Send it to downstream block
      ...
      end
  endtask

endclass
```

How do you exchange transactions between blocks? With procedural code you could have one object call the next, or you could use a data structure such as a FIFO to hold transactions in flight between blocks. In Chap. 7, you will learn how to use mailboxes, which are FIFOs with the ability to stall a thread until a new value is added.

5.19 Conclusion

Using Object-Oriented Programming is a big step, especially if your first computer language was Verilog. The payoff is that your testbenches are more modular and thus easier to develop, debug, and reuse.

Have patience – your first OOP testbench may look more like Verilog with a few classes added. As you get the hang of this new way of thinking, you begin to create and manipulate classes for both transactions and the transactors in the testbench that manipulate them.

In Chap. 8 you will learn more OOP techniques and so your test can change the behavior of the underlying testbench without having to change any of the existing code.

Chapter 6

Randomization

6.1 Introduction

As designs grow larger, it becomes more difficult to create a complete set of stimuli needed to check their functionality. You can write a directed test case to check a certain set of features, but you cannot write enough directed test cases when the number of features keeps doubling on each project. Worse yet, the interactions between all these features are the source for the most devious bugs and are the least likely to be caught by going through a laundry list of features.

The solution is to create test cases automatically using constrained-random tests (CRT). A directed test finds the bugs you think are there, but a CRT finds bugs you never thought about, by using random stimulus. You restrict the test scenarios to those that are both valid and of interest by using constraints.

Creating a CRT environment takes more work than creating one for directed tests. A simple directed test just applies stimulus, and then you manually check the result. These results are captured as a golden log file and compared with future simulations to see whether the test passes or fails. A CRT environment needs not only to create the stimulus but also to predict the result, using a reference model, transfer function, or other techniques. However, once this environment is in place, you can run hundreds of tests without having to hand-check the results, thereby improving your productivity. This trade-off of test-authoring time (your work) for CPU time (machine work) is what makes CRT so valuable.

A CRT is made of two parts: the test code that uses a stream of random values to create input to the DUT, and a seed to the pseudo-random number generator (PRNG), shown in Section 6.16.1. You can make a CRT behave differently just by using a new seed. This feature allows you to leverage each test so that each is the functional equivalent of many directed tests, just by changing seeds. You are able to create more equivalent tests using these techniques than with directed testing.

You may feel that these random tests are like throwing darts. How do you know when you have covered all aspects of the design? The stimulus space is too large to generate every possible input by using `for`-loops, and so you need to generate a useful subset. In Chap. 9 you will learn how to measure verification progress by using functional coverage.

There are many ways to use randomization, and this chapter gives a wide range of examples. It highlights the most useful techniques, but you should choose what works best for you.

6.2 What to Randomize

When you think of randomizing the stimulus to a design, the first thing you may think of are the data fields. These are the easiest to create – just call `$random`. The problem is that this approach has a very low payback in terms of bugs found: you only find data-path bugs, perhaps with bit-level mistakes. The test is still inherently directed. The challenging bugs are in the control logic. As a result, you need to randomize all decision points in your DUT. Wherever control paths diverge, randomization increases the probability that you'll take a different path in each test case.

You need to think broadly about all design input such as the following:

- Device configuration
- Environment configuration
- Primary input data
- Encapsulated input data
- Protocol exceptions
- Delays
- Transaction status
- Errors and violations

6.2.1 Device Configuration

What is the most common reason why bugs are missed during testing of the RTL design? Not enough different configurations have been tried! Most tests just use the design as it comes out of reset, or apply a fixed set of initialization vectors to put it

into a known state. This is like testing a PC's operating system right after it has been installed, and without any applications; of course the performance is fine, and there are no crashes.

Over time, in a real world environment, the DUT's configuration becomes more and more random. For example, a verification engineer had to verify a time-division multiplexor switch that had 600 input channels and 12 output channels. When the device was installed in the end-customer's system, channels would be allocated and deallocated over and over. At any point in time, there would be little correlation between adjacent channels. In other words, the configuration would seem random.

To test this device, the verification engineer had to write several dozen lines of Tcl code to configure each channel. As a result, she was never able to try configurations with more than a handful of channels enabled. Using a CRT methodology, she wrote a testbench that randomized the parameters for a single channel, and then put this in a loop to configure the whole device. Now she had confidence that her tests would uncover bugs that previously would have been missed.

6.2.2 Environment Configuration

The device that you are designing operates in an environment containing other devices. When you are verifying the DUT, it is connected to a testbench that mimics this environment. You should randomize the entire environment, including the number of objects and how they are configured.

Another company was creating an I/O switch chip that connected multiple PCI buses to an internal memory bus. At the start of simulation the customer used randomization to choose the number of PCI buses (1–4), the number of devices on each bus (1–8), and the parameters for each device (master or slave, CSR addresses, etc.). Even though there were many possible combinations, this company knew all had been covered.

6.2.3 Primary Input Data

This is what you probably thought of first when you read about random stimulus: take a transaction such as a bus write or ATM cell and fill it with some random values. How hard can that be? Actually it is fairly straightforward as long as you carefully prepare your transaction classes. You should anticipate any layered protocols and error injection.

6.2.4 Encapsulated Input Data

Many devices process multiple layers of stimulus. For example, a device may create TCP traffic that is then encoded in the IP protocol, and finally sent out inside Ethernet packets. Each level has its own control fields that can be randomized to try new combinations. So you are randomizing the data and the layers that surround it.

You need to write constraints that create valid control fields but that also allow injecting errors.

6.2.5 Protocol Exceptions, Errors, and Violations

Anything that can go wrong, will, eventually. The most challenging part of design and verification is how to handle errors in the system. You need to anticipate all the cases where things can go wrong, inject them into the system, and make sure the design handles them gracefully, without locking up or going into an illegal state. A good verification engineer tests the behavior of the design to the edge of the functional specification and sometimes even beyond.

When two devices communicate, what happens if the transfer stops partway through? Can your testbench simulate these breaks? If there are error detection and correction fields, you must make sure all combinations are tried.

The random component of these errors is that your testbench should be able to send functionally correct stimuli and then, with the flip of a configuration bit, start injecting random types of errors at random intervals.

6.2.6 Delays

Many communication protocols specify ranges of delays. The bus grant comes one to three cycles after request. Data from the memory is valid in the fourth to tenth bus cycle. However, many directed tests, optimized for the fastest simulation, use the shortest latency, except for that one test that only tries various delays. Your testbench should always use random, legal delays during every test to try to find that (hopefully) one combination that exposes a design bug.

Below the cycle level, some designs are sensitive to clock jitter. By sliding the clock edges back and forth by small amounts, you can make sure your design is not overly sensitive to small changes in the clock cycle.

The clock generator should be in a module outside the testbench so that it creates events in the Active region along with other design events. However, the generator should have parameters such as frequency and offset that can be set by the testbench during the configuration phase.

(Note that you are looking for functional errors, not timing errors. Your testbench should not try to violate setup and hold requirements. These are better validated using timing analysis tools.)

6.3 Randomization in SystemVerilog

The random stimulus generation in SystemVerilog is most useful when used with OOP. You first create a class to hold a group of related random variables, and then have the random-solver fill them with random values. You can create constraints to limit the random values to legal values, or to test-specific features.

Note that you can randomize individual variables, but this case is the least interesting. True constrained-random stimuli is created at the transaction level, not one value at a time.

6.3.1 Simple Class with Random Variables

Sample 6.1 shows a packet class with random variables and constraints, plus test-bench code that constructs and randomizes a packet.

Sample 6.1 Simple random class

```
class Packet;
  // The random variables
  rand bit [31:0] src, dst, data[8];
  randc bit [7:0] kind;
  // Limit the values for src
  constraint c {src > 10;
                src < 15;}
endclass

Packet p;
initial begin
  p = new();// Create a packet
  assert (p.randomize())
  else $fatal(0, "Packet::randomize failed");
  transmit(p);
end
```

This class has four random variables. The first three use the `rand` modifier, so that every time you randomize the class, the variables are assigned a value. Think of rolling dice: each roll could be a new value or repeat the current one. The `kind` variable is `randc`, which means random cyclic, so that the random solver does not repeat a random value until every possible value has been assigned. Think of dealing cards from a deck: you deal out every card in the deck in random order, then shuffle the deck, and deal out the cards in a different order. Note that the cyclic pattern is for a single variable. A `randc` array with eight elements has eight different patterns.

A constraint is just a set of relational expressions that must be true for the chosen value of the variables. In this example, the `src` variable must be greater than 10 and less than 15. Note that the constraint expression is grouped using curly braces: {}. This is because this code is declarative, not procedural, which uses `begin...end`.

The `randomize()` function returns 0 if a problem is found with the constraints. The procedural assertion is used to check the result, as shown in Section 4.8. This example uses a `$fatal` to stop simulation, but the rest of the book leaves out this extra code. You need to find the tool-specific switches to force the assertion to terminate simulation. This book uses `assert` to test the result from `randomize()`, but you may want to test the result, call your special routine that prints any useful information and then gracefully shut down the simulation.



You should not randomize an object in the class constructor. Your test may need to turn constraints on or off, change weights, or even add new constraints before randomization. The constructor is for initializing the object's variables, and if you called randomize() at this early stage, you might end up throwing away the results.



All variables in your classes should be random and public. This gives your test the maximum control over the DUT's stimulus and control. You can always turn off a random variable, as show in Section 6.11.2. If you forget to make a variable random, you must edit the environment, which you want to avoid.

6.3.2 Checking the Result from Randomization



The `randomize()` function assigns random values to any variable in the class that has been labeled as `rand` or `randc`, and also makes sure that all active constraints are obeyed. Randomization can fail if your code has conflicting constraints (see next section), and so you should always check the status. If you don't check, the variables may get unexpected values, causing your simulation to fail.

Sample 6.1 checks the status from `randomize()` by using a procedural assertion. If randomization succeeds, the function returns 1. If it fails, `randomize()` returns 0. The assertion checks the result and prints an error if there was a failure. You should set your simulator's switches to terminate when an error is found. Alternatively, you might want to call a special routine to end simulation, after doing some housekeeping chores like printing a summary report.

6.3.3 The Constraint Solver

The process of solving constraint expressions is handled by the SystemVerilog constraint solver. The solver chooses values that satisfy the constraints. The values come from SystemVerilog's PRNG, which is started with an initial seed. If you give a SystemVerilog simulator the same seed and the same testbench, it always produces the same results.

The solver is specific to the simulation vendor, and a constrained-random test may not give the same results when run on different simulators, or even on different versions of the same tool. The SystemVerilog standard specifies the meaning of the expressions, and the legal values that are created, but does not detail the precise order

in which the solver should operate. See Section 6.16 for more details on random number generators.

6.3.4 What can be Randomized?

SystemVerilog allows you to randomize integral variables, that is, variables that contain a simple set of bits. This includes 2-state and 4-state types, though randomization only works with 2-state values. You can have integers, bit vectors, etc. You cannot have a random string, or refer to a handle in a constraint.[1]

6.4 Constraint Details

Useful stimulus is more than just random values – there are relationships between the variables. Otherwise, it may take too long to generate interesting stimulus values, or the stimulus might contain illegal values. You define these interactions in SystemVerilog using constraint blocks that contain one or more constraint expressions. SystemVerilog chooses random values so that the expressions are true.



At least one variable in each expression should be random, either `rand` or `randc`. The following class fails when randomized, unless `age` happens to be in the right range. The solution is to add the modifier `rand` or `randc` before `age`.

Sample 6.2 Constraint without random variables

```
class Child;
  bit [31:0] age;  // Error Ð should be rand or randc
  constraint c_teenager {age > 12;
                         age < 20;}
endclass
```

The `randomize()` function tries to assign new values to random variables and to make sure all constraints are satisfied. In Sample 6.2, since there are no random variables, `randomize()` just checks the value of `son` to see if it is in the bounds specified by the constraint `c_teenager`. Unless the variable happens to fall in the range of 13:19, `randomize()` fails. While you can use a constraint to check that a nonrandom variable has a valid value, use an `assert` or `if`-statement instead. It is much easier to debug your procedural checker code than read through an error message from the random solver.

[1] As of late 2007, the IEEE SystemVerilog committee is still working on specifying how to randomize real variables. The issue is that the solver may not be able to solve a constraint such as one_third == 0.333 as the fraction 1/3 cannot be represented precisely as a real number.

6.4.1 Constraint Introduction

Sample 6.3 shows a simple class with random variables and constraints. The specific constructs are explained in the following sections.

Sample 6.3 Constrained-random class

```
class Stim;
  const bit [31:0] CONGEST_ADDR = 42;
  typedef enum {READ, WRITE, CONTROL} stim_e;
  randc stim_e kind;   // Enumerated var
  rand bit [31:0] len, src, dst;
  bit congestion_test;

  constraint c_stim {
    len < 1000;
    len > 0;
    if (congestion_test) {
      dst inside {[CONGEST_ADDR-100:CONGEST_ADDR+100]};
      src == CONGEST_ADDR;
    }
    else
      src inside {0, [2:10], [100:107]};
  }
endclass
```

6.4.2 Simple Expressions

Sample 6.3 showed a constraint block with several expressions. The first two control the values for the `len` variable. As you can see, a variable can be used in multiple expressions.



There can be a maximum of only one relational operator (<, <=, ==, >=, or >) in an expression. Sample 6.4 incorrectly tries to generate three variables in a fixed order.

Sample 6.4 Bad ordering constraint

```
class order;
  rand bit [7:0] lo, med, hi;
  constraint bad  {lo < med < hi;} // Gotcha!
endclass
```

Sample 6.5 Result from incorrect ordering constraint

```
lo =  20, med = 224, hi = 164
lo = 114, med =  39, hi = 189
lo = 186, med = 148, hi = 161
lo = 214, med = 223, hi = 201
```

Sample 6.5 shows the results, which are not what was intended. The constraint `bad` in Sample 6.4 is broken down into multiple binary relational expressions, going from left to right: `((lo < med) < hi)`. First, the expression `(lo < med)` is evaluated, which gives 0 or 1. Then `hi` is constrained to be greater than the result. The variables `lo` and `med` are randomized but not constrained. The correct constraint is shown in Sample 6.6. For more examples, see Sutherland and Mills (2007).

Sample 6.6 Constrain variables to be in a fixed order

```
class order;
  rand bit [15:0] lo, med, hi;
  constraint good {lo < med;   // Only use binary constraints
                   med < hi;}
endclass
```

6.4.3 Equivalence Expressions



The most common mistake with constraints is trying to make an assignment in a constraint block, but it can only contain expressions. Instead, use the equivalence operator to set a random variable to a value, e.g., `len==42`. You can build complex relationships between one or more random variables, such as `len == header.addr_mode * 4 + payload.size()`.

6.4.4 Weighted Distributions

The `dist` operator allows you to create weighted distributions so that some values are chosen more often than others. The `dist` operator takes a list of values and weights, separated by the `:=` or the `:/` operator. The values and weights can be constants or variables. The values can be a single value or a range such as `[lo:hi]`. The weights are not percentages and do not have to add up to 100. The `:=` operator specifies that the weight is the same for every specified value in the range, whereas the `:/` operator specifies that the weight is to be equally divided between all the values.

Sample 6.7 Weighted random distribution with `dist`

```
rand int src, dst;
constraint c_dist {
   src dist {0:=40, [1:3]:=60};
   // src = 0, weight = 40/220
   // src = 1, weight = 60/220
   // src = 2, weight = 60/220
   // src = 3, weight = 60/220

   dst dist {0:/40, [1:3]:/60};
   // dst = 0, weight = 40/100
   // dst = 1, weight = 20/100
   // dst = 2, weight = 20/100
   // dst = 3, weight = 20/100
}
```

In Sample 6.7, `src` gets the value 0, 1, 2, or 3. The weight of 0 is 40, whereas, 1, 2, and 3 each have the weight of 60, for a total of 220. The probability of choosing 0 is 40/220, and the probability of choosing 1, 2, or 3 is 60/220 each.

Next, `dst` gets the value 0, 1, 2, or 3. The weight of 0 is 40, whereas 1, 2, and 3 share a total weight of 60, for a total of 100. The probability of choosing 0 is 40/100, and the probability of choosing 1, 2, or 3 is only 20/100 each.

Once again, the values and weights can be constants or variables. You can use variable weights to change distributions on the fly or even to eliminate choices by setting the weight to zero, as shown in Sample 6.8.

Sample 6.8 Dynamically changing distribution weights

```
// Bus operation, byte, word, or longword
class BusOp;
  // Operand length
  typedef enum {BYTE, WORD, LWRD } length_e;
  rand length_e len;

  // Weights for dist constraint
  bit [31:0] w_byte=1, w_word=3, w_lwrd=5;

  constraint c_len {
    len dist {BYTE := w_byte,      // Choose a random
              WORD := w_word,      // length using
              LWRD := w_lwrd};     // variable weights
  }
endclass
```

In Sample 6.8, the `len` enumerated variable has three values. With the default weighting values, longword lengths are chosen more often, as `w_lwrd` has the largest value.

Don't worry, you can change the weights on the fly during simulation to get a different distribution.

6.4.5 Set Membership and the Inside Operator

You can create sets of values with the `inside` operator. The SystemVerilog solver chooses between the values in the set with equal probability, unless you have other constraints on the variable. As always, you can use variables in the sets.

Sample 6.9 Random sets of values

```
rand int c;             // Random variable
int lo, hi;             // Non-random variables used as limits
constraint c_range {
  c inside {[lo:hi]};  // lo <= c && c <= hi
}
```

In Sample 6.9, SystemVerilog uses the values for `lo` and `hi` to determine the range of possible values. You can use this to parameterize your constraints so that the testbench can alter the behavior of the stimulus generator without rewriting the constraints. Note that if `lo` > `hi`, an empty set is formed, and the constraint fails.

You can use `$` as a shortcut for the minimum and maximum values for a range, as shown in Sample 6.10. This is helpful when you are building constraints for variables with different ranges.

Sample 6.10 Specifying minimum and maximum range with $

```
rand bit [6:0] b;           // 0 <= b <= 127
rand bit [5:0] e;           // 0 <= e <= 63
constraint c_range {
  b inside {[$:4], [20:$]};  // 0 <= b <= 4 || 20 <= b <= 127
  e inside {[$:4], [20:$]};  // 0 <= e <= 4 || 20 <= e <= 63
}
```

If you want any value, as long as it is not inside a set, invert the constraint with the NOT operator: `!`

Sample 6.11 Inverted random set constraint

```
constraint c_range {
  !(c inside {[lo:hi]});  // c < lo or c > hi
}
```

6.4.6 Using an Array in a Set

You can choose from a set of values by storing them in an array.

Sample 6.12 Random set constraint for an array

```
rand int f;
int fib[5] = Õ{1,2,3,5,8};
constraint c_fibonacci {
  f inside fib;
}
```

This is expanded into the following set of constraints:

Sample 6.13 Equivalent set of constraints

```
constraint c_fibonacci {
  (f == fib[0]) ||   // f==1
  (f == fib[1]) ||   // f==2
  (f == fib[2]) ||   // f==3
  (f == fib[3]) ||   // f==5
  (f == fib[4]);     // f==8
}
```

All values in the set are chosen equally, even if they appear multiple times. You can also think of the `inside` constraint as being turned into a `foreach` constraint, as explained in Section 6.13.4.

Sample 6.14 chooses values using an `inside` constraint with repeated value, and also prints a histogram of the chosen values and so you can see that they are chosen equally.

Sample 6.14 Repeated values in inside constraint

```
class Weighted;
  rand int val;
  int array[] = '{1,1,2,3,5,8,8,8,8,8};
  constraint c {val inside array;}
endclass

Weighted w;
initial begin
  int count[9], maxx[$];
  w = new();

  repeat (2000) begin
    assert(w.randomize());
    count[w.val]++;         // Count the number of hits
   end

  maxx = count.max();     // Get largest value in count

  // Print histogram of count
  foreach(count[i])
  if (count[i]) begin
    $write("count[%0d]=%5d ", i, count[i]);
    repeat (count[i]*40/maxx[0]) $write("*");
    $display;
  end
end
```

Sample 6.15 Output from inside constraint operator and weighted array

```
count[1]= 3941 ************************************
count[2]= 4038 ****************************************
count[3]= 3978 ************************************
count[5]= 4027 ************************************
count[8]= 4016 ************************************
```

The right way to build a weighted distribution is with the `dist` operator as shown in Section 6.4.4.

Examples 6.16 and 6.17 choose a day of the week from a list of enumerated values. You can change the list of choices on the fly. If you make `choice` a `randc` variable, the simulator tries every possible value before repeating.

Sample 6.16 Class to choose from an array of possible values

```
class Days;
  typedef enum {SUN, MON, TUE, WED,
                THU, FRI, SAT} days_e;
  days_e choices[$];
  rand days_e choice;
  constraint cday {choice inside choices;}
endclass
```

Sample 6.17 Choosing from an array of values

```
initial begin
  Days days;
  days = new();

  days.choices = {Days::SUN, Days::SAT};
  assert (days.randomize());
  $display("Random weekend day %s\n", days.choice.name);

  days.choices = {Days::MON, Days::TUE, Days::WED,
                  Days::THU, Days::FRI};
  assert (days.randomize());
  $display("Random week day %s", days.choice.name);
end
```

The `name()` function returns a string with the name of an enumerated value.

If you want to dynamically add or remove values from a set, think twice before using the `inside` operator because of its performance. For example, perhaps you have a set of values that you want to choose just once. You could use `inside` to choose values from a queue, and delete them to slowly shrink the queue. This requires the solver to solve *N* constraints, where *N* is the number of elements left in the queue. Instead, use a `randc` variable that points into an array of choices. Choosing a `randc` value takes a short, constant time, whereas solving a large number of constraints is more expensive, especially if your array has more than a few dozen elements.

Sample 6.18 Using `randc` to choose array values in random order

```
class RandcInside;
  int array[];                  // Values to choose
  randc bit [15:0] index;       // Index into array

  function new(input int a[]); // Construct & initialize
    array = a;
  endfunction

  function int pick;            // Return most recent pick
    return array[index];
  endfunction

  constraint c_size {index < array.size();}
endclass

  initial begin
    RandcInside ri;

    ri = new(õ{1,3,5,7,9,11,13});
    repeat (ri.array.size()) begin
      assert(ri.randomize());
      $display("Picked %2d [%0d]", ri.pick(),  ri.index);
    end
  end
```

Note that constraints and routines can be mixed in any order.

6.4.7 Conditional Constraints

Normally, all constraint expressions are active in a block. What if you want to have an expression active only some of the time? For example, a bus supports byte, word, and longword reads, but only longword writes. SystemVerilog supports two implication operators, `->` and `if-else`.

When you are choosing from a list of expressions, such as an enumerated type, the implication operator, `->`, lets you create a case-like block. The parentheses around the expression are not required, but do make the code easier to read.

Sample 6.19 Constraint block with implication operator

```
class BusOp;
  ...
  constraint c_io {
    (io_space_mode) ->
      addr[31] == 1'b1;
  }
```

If you have a true-false expression, the `if-else` operator may be better.

Sample 6.20 Constraint block with `if-else` operator

```
class BusOp;
  ...
  constraint c_len_rw {
    if (op == READ)
      len inside {[BYTE:LWRD]};
    else
      len == LWRD;
  }
```

In constraint blocks, you use curly braces, { }, to group multiple expressions. The `begin...end` keywords are for procedural code.

6.4.8 Bidirectional Constraints

By now you may have realized that constraint blocks are not procedural code, executing from top to bottom. They are declarative code, all active at the same time. If you constrain a variable with the `inside` operator with the set `[10:50]` and have another expression that constrains the variable to be greater than 20, SystemVerilog solves both constraints simultaneously and only chooses values between 21 and 50.

SystemVerilog constraints are bidirectional, which means that the constraints on all random variables are solved concurrently. Adding or removing a constraint on any one variable affects the value chosen for all variables that are related directly or indirectly. Consider the constraint in Sample 6.21.

Sample 6.21 Bidirectional constraint

```
rand logic [15:0] r, s, t;
constraint c_bidir {
  r < t;
  s == r;
  t < 30;
  s > 25;
}
```

The SystemVerilog solver looks at all four constraints simultaneously. The variable `r` has to be less than `t`, which has to be less than 30. However, `r` is also constrained to be equal to `s`, which is greater than 25. Even though there is no direct constraint on the lower value of `t`, the constraint on `s` restricts the choices. Table 6-1 shows the possible values for these three variables.

Table 6-1 Solutions for bidirectional constraint

Solution	r	s	t
A	26	26	27
B	26	26	28
C	26	26	29
D	27	27	28
E	27	27	29
F	28	28	29

Even the conditional constraints such as `->` and `if...else`, which can look like a procedural `if-else` statement, are bidirectional. For example, the constraint `{(a==1) -> (b==0)}` is equivalent to `{!(a == 1) || b == 0;}`. The solver picks values for the variables that meet this constraint, and does not first check if `a==1`, then force `b==0`. In fact, if you add the additional constraint `{b==1;}`, the solver will set `a` to 0.

6.4.9 Choose the Right Arithmetic Operator to Boost Efficiency

Simple arithmetic operators such as addition and subtraction, bit extracts, and shifts are handled very efficiently by the solver in a constraint. However, multiplication, division, and modulo are very expensive with 32-bit values. Remember that any constant without an explicit size, such as `42`, is treated as a 32-bit value.

If you want to generate random addresses that are near a page boundary, where a page is 4096 bytes, you could write the following code, but the solver may take a long time to find suitable values for `addr`.

Sample 6.22 Expensive constraint with mod and unsized variable

```
rand bit [31:0] addr;
constraint slow_near_page_boundary {
  addr % 4096 inside {[0:20], [4075:4095]};
}
```

Many constants in hardware are powers of 2, and so take advantage of this by using bit extraction rather than division and modulo. Likewise, multiplication by a power of two can be replaced by a shift.

Sample 6.23 Efficient constraint with bit extract

```
rand bit [31:0] addr;
constraint near_page_boundry {
  addr[11:0] inside {[0:20], [4075:4095]};
}
```

6.5 Solution Probabilities

Whenever you deal with random values, you need to understand the probability of the outcome. SystemVerilog does not guarantee the exact solution found by the random constraint solver, but you can influence the distribution. Any time you work with random numbers, you have to look at thousands or millions of values to average out the noise. Changing the tool version or random seed can cause different results. Some simulators, such as Synopsys VCS, have multiple solvers to allow you to trade memory usage vs. performance.

6.5.1 Unconstrained

Start with two variables with no constraints.

Sample 6.24 Class `Unconstrained`

```
class Unconstrained;
  rand bit x;          // 0 or 1
  rand bit [1:0] y;    // 0, 1, 2, or 3
endclass
```

There are eight possible solutions, as shown in Table 6-2. Since there are no constraints, each has the same probability. You have to run thousands of randomizations to see the actual results that approach the listed probabilities.[2]

Table 6-2 Solutions for `Unconstrained` class

Solution	x	y	Probability
A	0	0	1/8
B	0	1	1/8
C	0	2	1/8
D	0	3	1/8
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

[2]The tables were generated with Synopsys VCS 2005.06 using the run-time switch `+ntb_solver_mode=1`.

6.5.2 Implication

In Sample 6.25, the value of `y` depends on the value of `x`. This is indicated with the implication operator in the following constraint. This example and the rest in this section also behave in the way same with the `if` implication operator.

Sample 6.25 Class with implication

```
class Imp1;
  rand bit x;          // 0 or 1
  rand bit [1:0] y;    // 0, 1, 2, or 3
  constraint c_xy {
    (x==0) -> y==0;
  }
endclass
```

Here are the possible solutions and probability. You can see that the random solver recognizes that there are eight combinations of `x` and `y`, but all the solutions where `x==0` (A–D) have been merged together (Table 6-3).

Table 6-3 Solutions for `Imp1` class

Solution	x	y	Probability
A	0	0	1/2
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

6.5.3 Implication and Bidirectional Constraints

Note that the implication operator says that when `x==0`, `y` is forced to 0, but when `y==0`, there is no constraint on `x`. However, implication is bidirectional in that if `y` were forced to a nonzero value, `x` would have to be 1. Sample 6.26 has the constraint `y>0`, and so `x` can never be 0 (Table 6-4).

Sample 6.26 Class with implication and constraint

```
class Imp2;
  rand bit x;          // 0 or 1
  rand bit [1:0] y;    // 0, 1, 2, or 3
  constraint c_xy {
     y > 0;
    (x==0) -> y==0;
  }
endclass
```

Table 6-4 Solutions for `Imp2` class

Solution	x	y	Probability
A	0	0	0
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	0
F	1	1	1/3
G	1	2	1/3
H	1	3	1/3

6.5.4 Guiding Distribution with `solve...before`

You can guide the SystemVerilog solver using the "`solve...before`" constraint as seen in Sample 6.27.

Sample 6.27 Class with implication and `solve...before`

```
class SolveBefore;
  rand bit x;          // 0 or 1
  rand bit [1:0] y;    // 0, 1, 2, or 3
  constraint c_xy {
    (x==0) -> y==0;
    solve x before y;
  }
endclass
```

The `solve...before` constraint does not change the solution space, just the probability of the results. The solver chooses values of `x` (0, 1) with equal probability. In 1,000 calls to `randomize()`, `x` is 0 about 500 times, and 1 about 500 times. When `x` is 0, `y` must be 0. When `x` is 1, `y` can be 0, 1, 2, or 3 with equal probability (Table 6-5).

Table 6-5 Solutions for `solve x before y` constraint

Solution	x	y	Probability
A	0	0	1/2
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

However, if you use the constraint `solve y before x`, you get a very different distribution (Table 6-6).

Table 6-6 Solutions for `solve y before x` constraint

Solution	x	y	Probability
A	0	0	1/8
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/4
G	1	2	1/4
H	1	3	1/4



Only use `solve...before` if you are dissatisfied with how often some values occur. Excessive use can slow the constraint solver and make your constraints difficult for others to understand.

6.6 Controlling Multiple Constraint Blocks

A class can contain multiple constraint blocks. One might make sure you have a valid transaction, as described in Section 6.7, but you might need to disable this when testing the DUT's error handling. Or you might want to have a separate constraint for each test. Perhaps one constraint would restrict the data length to create small transactions (great for testing congestion), whereas another would make long transactions.

At run-time, you can use the built-in `constraint_mode()` routine to turn constraints on and off. You can control a single constraint with `handle.constraint.constraint_mode()`. To control all constraints in an object, use `handle.constraint_mode()`, as shown in Sample 6.28.

Sample 6.28 Using `constraint_mode`

```
class Packet;
  rand int length;
  constraint c_short {length inside {[1:32]}; }
  constraint c_long  {length inside {[1000:1023]}; }
endclass

Packet p;
initial begin
  p = new();

  // Create a long packet by disabling short constraint
  p.c_short.constraint_mode(0);
  assert (p.randomize());

  transmit(p);

  // Create a short packet by disabling all constraints
  // then enabling only the short constraint
  p.constraint_mode(0);
  p.c_short.constraint_mode(1);
  assert (p.randomize());
  transmit(p);
end
```

6.7 Valid Constraints

A good randomization technique is to create several constraints to ensure the correctness of your random stimulus, known as "valid constraints." For example, a bus read–modify–write command might only be allowed for a longword data length.

Sample 6.29 Checking write length with a valid constraint

```
class Transaction;
  rand enum {BYTE, WORD, LWRD, QWRD} length;
  rand enum {READ, WRITE, RMW, INTR} opc;

  constraint valid_RMW_LWRD {
    (opc == RMW) -> length == LWRD;
  }
endclass
```

Now you know the bus transaction obeys the rule. Later, if you want to violate the rule, use `constraint_mode` to turn off this one constraint. You should have a

naming convention to make these constraints stand out, such as using the prefix `valid` as shown above.

6.8 In-line Constraints

As you write more tests, you can end up with many constraints. They can interact with each other in unexpected ways, and the extra code to enable and disable them adds to the test complexity. Additionally, constantly adding and editing constraints to a class could cause problems in a team environment.

Many tests only randomize objects at one place in the code. SystemVerilog allows you to add an extra constraint using `randomize() with`. This is equivalent to adding an extra constraint to any existing ones in effect. Sample 6.30 shows a base class with constraints, then two `randomize() with` statements.

Sample 6.30 The `randomize() with` statement

```
class Transaction;
  rand bit [31:0] addr, data;
  constraint c1 {addr inside{[0:100],[1000:2000]};}
endclass

Transaction t;

initial begin
  t = new();

  // addr is 50-100, 1000-1500, data < 10
  assert(t.randomize() with {addr >= 50; addr <= 1500;
                             data < 10;});

  driveBus(t);

  // force addr to a specific value, data > 10
  assert(t.randomize() with {addr == 2000; data > 10;});

  driveBus(t);
  end
```

The extra constraints are added to the existing ones in effect. Use `constraint_mode` if you need to disable a conflicting constraint. Note that inside the `with{}` statement, SystemVerilog uses the scope of the class. That is why Sample 6.30 used just `addr`, not `t.addr`.



A common mistake is to surround your in-line constraints with parenthesis instead of curly braces {}. Just remember that constraint blocks use curly braces, and so your in-line constraint must use them too. Braces are for declarative code.

6.9 The `pre_randomize` and `post_randomize` Functions

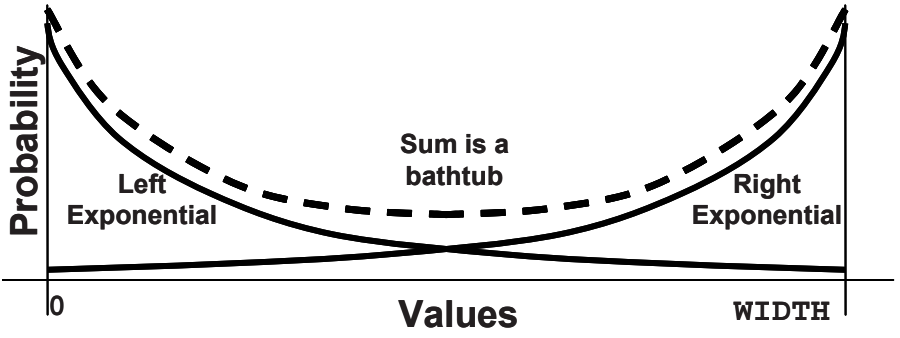
Sometimes you need to perform an action immediately before every `randomize()` call or immediately afterwards. For example, you may want to set some nonrandom class variables (such as limits or weights) before randomization starts, or you may need to calculate the error correction bits for random data.

SystemVerilog lets you do this with two special void functions, `pre_randomize` and `post_randomize`. Section 3.2 showed that a void function does not return a value, but, because it is not a task, does not consume time. If you want to call a debug routine from `pre_randomize` or `post_randomize`, it must be a function.

6.9.1 Building a Bathtub Distribution

For some applications, you want a nonlinear random distribution. For instance, small and large packets are more likely to find a design bug such as buffer overflow than medium-sized packets. So you want a bathtub shaped distribution; high on both ends, and low in the middle. You could build an elaborate `dist` constraint, but it might require lots of tweaking to get the shape you want. Verilog has several functions for nonlinear distribution such as `$dist_exponential` but none for a bathtub. The graph in Figure 6.1 shows how you can combine two exponential curves to make a bathtub curve. The `pre_randomize` method in Sample 6.31 calculates a point on an exponential curve, and then randomly chooses to put this on the left curve, or right. As you pick points on either the left and right curves, you gradually build a distribution of the combined values.

Figure 6-1 Building a bathtub distribution



Sample 6.31 Building a bathtub distribution

```
class Bathtub;
  int value;  // Random variable with bathtub dist
  int WIDTH = 50, DEPTH=4, seed=1;

  function void pre_randomize();
    // Calculate an exponental curve
    value = $dist_exponential(seed, DEPTH);
    if (value > WIDTH) value = WIDTH;

    // Randomly put this point on the left or right curve
    if ($urandom_range(1))
      value = WIDTH - value;
  endfunction

endclass
```

Every time this object is randomized, the variable `value` gets updated. Across many randomizations, you will see the desired nonlinear distribution. Since the variable is calculated procedurally, not through the random constraint solver, it does not need the `rand` modifier.

6.9.2 Note on Void Functions



The functions `pre_randomize` and `post_randomize` can only call other functions, not tasks that could consume time. After all, you cannot have a delay in the middle of a call to `randomize()`. When you are debugging a randomization problem, you can call your display routines if you planned ahead and made them void functions.

6.10 Random Number Functions

You can use all the Verilog-1995 distribution functions, plus several that are new for SystemVerilog. Consult a statistics book for more details on the "dist" functions. Some of the useful functions include the following.

- `$random`() – Flat distribution, returning signed 32-bit random
- `$urandom`() – Flat distribution, returning unsigned 32-bit random
- `$urandom_range`() – Flat distribution over a range
- `$dist_exponential()` – Exponential decay, as shown in Figure 6-1
- `$dist_normal()` – Bell-shaped distribution
- `$dist_poisson()` – Bell-shaped distribution
- `$dist_uniform()` – Flat distribution

The `$urandom_range()` function takes two arguments, an optional low value, and a high value.

Sample 6.32 $urandom_range usage

```
a = $urandom_range(3, 10); // Pick a value from 3 to 10
a = $urandom_range(10, 3); // Pick a value from 3 to 10
b = $urandom_range(5);     // Pick a value from 0 to 5
```

6.11 Constraints Tips and Techniques

How can you create CRT that can be easily modified? There are several tricks you can use. The most general technique is to use OOP to extend the original class as described in Sections 6.11.8 and 8.2.4, but this also requires more planning. So, first learn some simple techniques, but keep an open mind.

6.11.1 Constraints with Variables

Most constraint examples in this book use constants to make them more readable. In Sample 6.33, `size` is randomized over a range that uses a variable for the upper bound.

Sample 6.33 Constraint with a variable bound

```
class bounds;
  rand int size;
  int max_size = 100;
  constraint c_size {
   size inside {[1:max_size]};
  }
endclass
```

By default, this class creates random sizes between 1 and 100, but by changing the variable `max_size`, you can vary the upper limit.

You can use variables in the `dist` constraint to turn on and off values and ranges. In Sample 6.34, each bus command has a different weight variable.

Sample 6.34 `dist` constraint with variable weights

```
typedef enum (READ8, READ16, READ32) read_e;
class ReadCommands;
  rand read_e read_cmd;
  int read8_wt=1, read16_wt=1, read32_wt=1;
  constraint c_read {
    read_cmd dist {READ8  := read8_wt,
                   READ16 := read16_wt,
                   READ32 := read32_wt};
    }
endclass
```

By default, this constraint produces each command with equal probability. If you want to have a greater number of `READ8` commands, increase the `read8_wt` weight variable. Most importantly, you can turn off generation of some commands by dropping their weight to 0.

6.11.2 Using Nonrandom Values

If you have a set of constraints that produces stimulus that is almost what you want, but not quite, you could call `randomize()`, and then set a variable to the value you want – you don't have to use the random one. However, your stimulus values may not be correct according to the constraints you created to check validity.

If there are just a few variables that you want to override, use `rand_mode` to make them nonrandom.

Sample 6.35 `rand_mode` disables randomization of variables

```
// Packet with variable length payload
class Packet;
  rand bit [7:0] length, payload[];
  constraint c_valid {length > 0;
                      payload.size() == length;}

  function void display(string msg);
    $display("\n%s", msg);
    $write("Packet len=%0d, payload size=%0d, bytes = ",
           length, payload.size());
    for(int i=0; (i<4 && i<payload.size()); i++)
      $write(" %0d", payload[i]);
    $display;
  endfunction
endclass

Packet p;
initial begin
  p = new();

  // Randomize all variables
  assert (p.randomize());
  p.display("Simple randomize");

  p.length.rand_mode(0);   // Make length nonrandom,
  p.length = 42;           // set it to a constant value
  assert (p.randomize());  // then randomize the payload
  p.display("Randomize with rand_mode");
end
```

In Sample 6.35, the packet size is stored in the random variable `length`. The first half of the test randomizes both the `length` variable and the contents of the `payload` dynamic array. The second half calls `rand_mode` to make `length` a nonrandom variable, sets it to 42, and then calls `randomize()`. The constraint sets the `payload` size at the constant 42, but the array is still filled with random values.

6.11.3 Checking Values Using Constraints

If you randomize an object and then modify some variables, you can check that the object is still valid by checking if all constraints are still obeyed. Call `handle.randomize(null)` and SystemVerilog treats all variables as nonrandom ("state variables") and just ensures that all constraints are satisfied.

6.11.4 Randomizing Individual Variables

Suppose you want to randomize a few variables inside a class. You can call `randomize()` with the subset of variables. Only those variables passed in the argument list will be randomized; the rest will be treated as state variables and not randomized. All constraints remain in effect. In Sample 6.36, the first call to `randomize()` only changes the values of two `rand` variables `med` and `hi`. The second call only changes the value of `med`, whereas `hi` retains its previous value. Surprisingly, you can pass a nonrandom variable, as shown in the last call, and `low` is given a random value, as long as it obeys the constraint.

Sample 6.36 Randomizing a subset of variables in a class

```
class Rising;
  byte low;            // Not random
  rand byte med, hi;  // Random variable
  constraint up
    { low < med; med < hi; } // See Section 6.4.2
endclass

initial begin
  Rising r;
  r = new();
  r.randomize();     // Randomize med, hi; low untouched
  r.randomize(med);  // Randomize only med
  r.randomize(low);  // Randomize only low
end
```

This trick of only randomizing a subset of the variables is not commonly used in real testbenches as you are restricting the randomness of your stimulus. You want your testbench to explore the full range of legal values, not just a few corners.

6.11.5 Turn Constraints Off and On

A simple testbench may use a data class with just a few constraints. What if you want to have two tests with very different flavors of data? You could use the implication operators (`->` or `if-else`) to build a single, elaborate constraint controlled by nonrandom variables.

Sample 6.37 Using the implication constraint as a `case` statement

```
class Instruction;
  typedef enum {NOP, HALT, CLR, NOT} opcode_e;
  rand opcode_e opcode;
  bit [1:0] n_operands;
  ...
  constraint c_operands {
    if (n_operands == 0)
      opcode == NOP || opcode == HALT;
    else if (n_operands == 1)
      opcode == CLR || opcode == NOT;
    ...
  }
endclass
```

You can see that having one large constraint can quickly get out of control as you add further expressions for each operand, addressing modes, etc. A more modular approach is to use a separate constraint for each flavor of instruction, and then disable all but the one you need, as shown in Sample 6.38.

Sample 6.38 Turning constraints on and off with `constraint_mode`

```
class Instruction;
  rand opcode_e opcode;
  É
  constraint c_no_operands {
    opcode == NOP || opcode == HALT;}
  constraint c_one_operand {
    opcode == CLR || opcode == NOT;}
endclass

Instruction instr;
initial begin
  instr = new();

  // Generate an instruction with no operands
  instr.constraint_mode(0);  // Turn off all constraints
  instr.c_no_operands.constraint_mode(1);
  assert (instr.randomize());

  // Generate an instruction with one operand
  instr.constraint_mode(0);  // Turn off all constraints
  instr.c_one_operand.constraint_mode(1);
  assert (instr.randomize());
  end
```

While many small constraints may give you more flexibility, the process of turning them on and off is more complex. For example, when you turn off all constraints that create data, you are also disabling all the ones that check the data's validity.

6.11.6 Specifying a Constraint in a Test Using In-Line Constraints

If you keep adding constraints to a class, it becomes hard to manage and control. Soon, everyone is checking out the same file from your source control system. Many times a constraint is only used by a single test, and so why have it visible to every test? One way to localize the effects of a constraint is to use in-line constraints, `randomize() with`, shown in Section 6.8. This works well if your new constraint is additive to the default constraints. If you follow the recommendations in Section 6.7 to create "valid constraints," you can quickly constrain valid sequences. For error injection, you can disable any constraint that conflicts with what you are trying to do. For example, if a test needs to inject a particular flavor of corrupted data, it would first turn off the particular validity constraint that checks for that error.

There are several tradeoffs with using in-line constraints. The first is that now your constraints are in multiple locations. If you add a new constraint to the original class, it may conflict with the in-line constraint. The second is that it can be very hard for you to reuse an in-line constraint across multiple tests. By definition, an in-line constraint only exists in one piece of code. You could put it in a routine in a separate file and then call it as needed. At that point it has become nearly the same as an external constraint.

6.11.7 Specifying a Constraint in a Test with External Constraints

The body of a constraint does not have to be defined within the class, just as a routine body can be defined externally, as shown in Section 5.11. Your data class could be defined in one file, with one empty constraint. Then each test could define its own version of this constraint to generate its own flavors of stimulus.

Sample 6.39 Class with an external constraint

```
// packet.sv
class Packet;
  rand bit [7:0] length;
  rand bit [7:0] payload[];
  constraint c_valid {length > 0;
                      payload.size() == length;}
  constraint c_external;
endclass
```

Sample 6.40 Program defining an external constraint

```
// test.sv
program test;
  constraint Packet::c_external {length == 1;}
  ...
endprogram
```

External constraints have several advantages over in-line constraints. They can be put in a file and thus reused between tests. An external constraint applies to all instances of the class, whereas an in-line constraint only affects the single call to `randomize()`. Consequently, an external constraint provides a primitive way to change a class without having to learn advanced OOP techniques. Keep in mind that with this technique, you can only add constraints, not alter existing ones, and you need to define the external constraint prototype in the original class.

Like in-line constraints, external constraints can cause problems, as the constraints are spread across multiple files.

A final consideration is what happens when the body for an external constraint is never defined. The SystemVerilog LRM does not currently specify what should happen in this case. Before you build a testbench with many external constraints, find out how your simulator handles missing definitions. Is this an error that prevents simulation, just a warning, or no message at all?

6.11.8 Extending a Class

In Chap. 8, you will learn how to extend a class. With this technique, you can take a testbench that uses a given class, and swap in an extended class that has additional or redefined constraints, routines, and variables. See Sample 8.10 for a typical testbench. Note that if you define a constraint in an extended class with the same name as one in the base class, the extended constraint replaces the base one.

Learning OOP techniques requires a little more study, but the flexibility of this new approach repays with great rewards.

6.12 Common Randomization Problems

You may be comfortable with procedural code, but writing constraints and understanding random distributions requires a new way of thinking. Here are some issues you may encounter when trying to create random stimulus.

6.12.1 Use Signed Variables with Care

When creating a testbench, you may be tempted to use the `int`, `byte`, or other signed types for counters and other simple variables. Don't use them in random constraints unless you really want signed values. What values are produced when the class in Sample 6.41 is randomized? It has two random variables and wants to make the sum of them 64.

Sample 6.41 Signed variables cause randomization problems

```
class SignedVars;
  rand byte pkt1_len, pk2_len;
  constraint total_len {
    pkt1_len + pk2_len == 64;
  }
endclass
```

Obviously, you could get pairs of values such as (32, 32) and (2, 62). Additionally, you could see (–63, 127), as this is a legitimate solution of the equation, even though it may not be what you wanted. To avoid meaningless values such as negative lengths, use only unsigned random variables, as shown in Sample 6.42.

Sample 6.42 Randomizing unsigned 32-bit variables

```
class Vars32;
  rand bit [31:0] pkt1_len, pk2_len;  // unsigned type
  constraint total_len {
    pkt1_len + pk2_len == 64;
  }
endclass
```

Even this version causes problems, as large values of `pkt1_len` and `pkt2_len`, such as `32'h80000040` and `32'h80000000`, wrap around when added together and give `32'd64` or `32'h40`. You might think of adding another pair of constraints to restrict the values of these two variables, but the best approach is to make them only as wide as needed, and to avoid using 32-bit variables in constraints. In Sample 6.43, the sum of two 8-bit variables is compared to a 9-bit value.

Sample 6.43 Randomizing unsigned 8-bit variables

```
class Vars8;
  rand bit [7:0] pkt1_len, pkt2_len;  // 8-bits wide
  constraint total_len {
    pkt1_len + pkt2_len == 9'd64;       // 9-bit sum
  }
endclass
```

6.12.2 Solver Performance Tips

Each constraint solver has its strengths and weaknesses, but there are some guidelines that you can follow to improve the speed of your simulations with constrained random variables.

Avoid expensive operators such as division, multiplication, and modulus (`%`). If you need to divide or multiply a variable by a power of 2, use the right and left shift operators. A modulus operation with a power of 2 can be replaced with boolean AND with a mask. If you need to use one of these operators, you may get better performance if you can use variables less than 32-bits wide.

6.13 Iterative and Array Constraints

The constraints presented so far allow you to specify limits on scalar variables. What if you want to randomize an array? The `foreach` constraint and several array functions let you shape the distribution of the values.



Using the `foreach` constraint creates many constraints that can slow down simulation. A good solver can quickly solve hundreds of constraints but may slow down with thousands. Especially slow are nested `foreach` constraints, as they produce N^2 constraints for an array of size N. See Section 6.13.5 for an algorithm that used `randc` variables instead of nested `foreach`.

6.13.1 Array Size

The easiest array constraint to understand is the `size()` function. You are specifying the number of elements in a dynamic array or queue.

Sample 6.44 Constraining dynamic array size

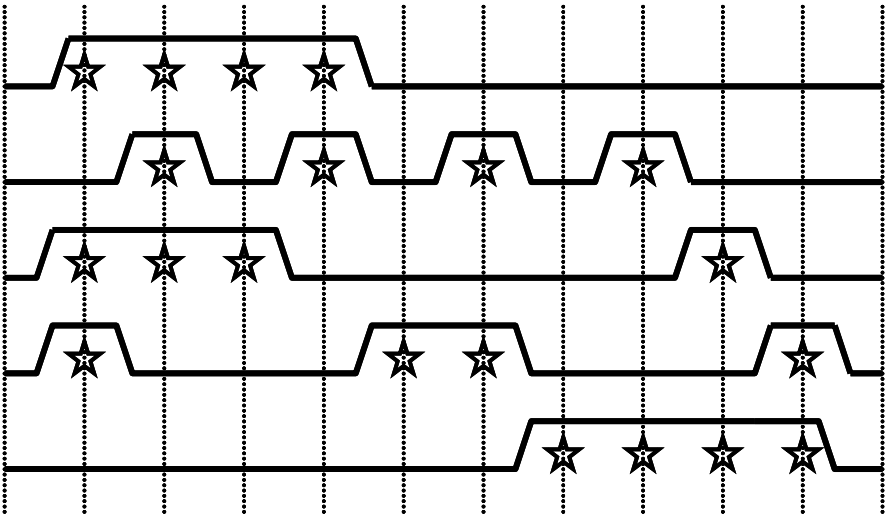
```
class dyn_size;
  rand logic [31:0] d[];
  constraint d_size {d.size() inside {[1:10]}; }
endclass
```

Using the `inside` constraint lets you set a lower and upper boundary on the array size. In many cases you may not want an empty array, that is, `size==0`. Remember to specify an upper limit; otherwise, you can end up with thousands or millions of elements, which can cause the random solver to take an excessive amount of time.

6.13.2 Sum of Elements

You can send a random array of data into a design, but you can also use it to control the flow. Perhaps you have an interface that has to transfer four data words. The words can be sent consecutively or over many cycles. A strobe signal tells when the data signal is valid. Figure 6-2 shows some legitimate strobe patterns, sending four values over ten cycles.

Figure 6-2 Random strobe waveforms



You can create these patterns using a random array. Constrain it to have four bits enabled out of the entire range using the `sum()` function.

Sample 6.45 Random strobe pattern class

```
parameter MAX_TRANSFER_LEN = 10;

class StrobePat;
  rand bit strobe[MAX_TRANSFER_LEN];
  constraint c_set_four { strobe.sum() == 4'h4; }
endclass

initial begin
  StrobePat sp;
  int count = 0;       // Index into data array

  sp = new();
  assert (sp.randomize());

  foreach (sp.strobe[i]) begin
    @bus.cb;
    bus.cb.strobe <= sp.strobe[i];
    // If strobe is enabled, drive out next data word
    if (sp.strobe[i])
      bus.cb.data <= data[count++];
  end
end
```

As you remember from Chap. 2, the sum of an array of single-bit elements would normally be a single bit, e.g., 0 or 1. Sample 6.45 compares `strobe.sum()` to a 4-bit value (`4'h4`), and so the sum is calculated with 4-bit precision. The example uses 4-bit precision to store the maximum number of elements, which is 10.

6.13.3 Issues with Array Constraints

The `sum()` function looks simple but can cause several problems because of Verilog's arithmetic rules. Start with a simple problem. You want to generate from one to eight transactions, such that the total length of all of them is less than 1,024 bytes. Sample 6.46 shows a first attempt. The `len` field is a byte in the original transaction.

Sample 6.46 First attempt at sum constraint: `bad_sum1`

```
class bad_sum1;
  rand byte len[];
  constraint c_len {len.sum < 1024;
                    len.size() inside {[1:8]};}
  function void display();
    $write("sum=%4d, val=", len.sum);
    foreach(len[i]) $write("%4d ", len[i]);
    $display;
  endfunction
endclass
```

Sample 6.47 Program to try constraint with array sum

```
program automatic test;
  bad_sum1 c;
  initial begin
    c = new();
    repeat (10) begin
      assert (c.randomize());
      c.display();
    end
  end
endprogram
```

Sample 6.48 Output from `bad_sum1`

```
sum=  81, val=  62 -20  39
sum=  39, val= -27  67   1  76 -97 -58  77
sum=  38, val=  60 -22
sum=  72, val=-120  29 123 102 -41 -21
sum= -53, val= -58 -85-115 112-101 -62
```

This generates some smaller lengths, but the sum is sometimes negative and is always less than 127 – definitely not what you wanted! Try again, this time with an unsigned field. (The `display` function is unchanged.)

Sample 6.49 Second attempt at sum constraint: `bad_sum2`

```
class bad_sum2;
  rand bit [7:0] len[];  // 8 bits
  constraint c_len {len.sum < 1024;
                    len.size() inside {[1:8]};}
endclass
```

Sample 6.50 Output from `bad_sum2`

```
sum=  79, val=  88 100 246   2  14 228 169
sum= 120, val=  74  75 141  86
sum=  39, val=  39
sum= 193, val=  31 156 172  33  57
sum= 173, val=  59 150  25 101 138 212
```

Sample 6.50 has a subtle problem: the sum of all transaction lengths is always less than 256, even though you constrained the array sum to be less than 1,024. The problem here is that in Verilog, the sum of many 8-bit values is computed using an 8-bit result. Bump the `len` field up to 32 bits using the `uint` type from Chap. 2.

Sample 6.51 Third attempt at sum constraint: `bad_sum3`

```
class bad_sum3;
  rand uint len[];  // 32 bits
  constraint c_len {len.sum < 1024;
                    len.size() inside {[1:8]};}
endclass
```

Sample 6.52 Output from `bad_sum3`

```
sum= 245, val=1348956995 3748256598 985546882 2507174362
sum= 600, val=2072193829 315191491 484497976 3050698208
 2300168220 3988671456 3998079060 970369544
sum=  17, val=1924767007 3550820640 4149215303 3260098955
sum= 440, val=3192781444 624830067 1300652226 4072252356
 3694386235
sum= 864, val=3561488468 733479692
```

Wow – what happened here? This is similar to the signed problem in Section 6.12.1, in that the sum of two very large numbers can wrap around to a small number. You need to limit the size based on the comparison in the constraint.

Sample 6.53 Fourth attempt at sum_constraint: `bad_sum4`

```
class bad_sum4;
  rand bit [9:0] len[];  // 10 bits
  constraint c_len {len.sum < 1024;
                    len.size() inside {[1:8]};}
endclass
```

Sample 6.54 Output from `bad_sum4`

```
sum= 989, val= 787 202
sum=1021, val= 564  76 132 235   0   8   6
sum= 872, val= 624 101 136  11
sum= 978, val= 890  88
sum= 905, val= 663 242
```

This does not work either, as the individual `len` fields are more than 8 bits, and so the `len` values are often greater than 255. You need to specify that each `len` field is between 1 and 255, but use a 10-bit field so that they sum correctly. This requires constraining every element of the array.

6.13.4 Constraining Individual Array and Queue Elements

SystemVerilog lets you constrain individual elements of an array using `foreach`. While you might be able to write constraints for a fixed-size array by listing every element, the `foreach` style is more compact. The only practical way to constrain a dynamic array or queue is with `foreach`.

Sample 6.55 Simple `foreach` constraint: `good_sum5`

```
class good_sum5;
  rand uint len[];
  constraint c_len {foreach (len[i])
                      len[i] inside {[1:255]};
                    len.sum < 1024;
                    len.size() inside {[1:8]};}
endclass
```

Sample 6.56 Output from `good_sum5`

```
sum=1011, val=  83 249 197 187 152  95  40   8
sum=1012, val= 213 252 213  44 196  20  20  54
sum= 370, val= 118  76 176
sum= 976, val= 233 187  44 157 201  81  73
sum= 412, val= 172 167  73
```

The addition of the constraint for individual elements fixed the example. Note that the `len` array can be 10 or more bits wide, but must be unsigned.

You can specify constraints between array elements as long as you are careful about the endpoints. The following class creates an ascending list of values by comparing each element to the previous, except for the first.

Sample 6.57 Creating ascending array values with `foreach`

```
class Ascend;
  rand uint d[10];
  constraint c {
    foreach (d[i])      // For every element
      if (i>0)          // except the first
        d[i] > d[i-1]; // compare with previous element
  }
endclass
```

How complex can these constraints become? Constraints have been written to solve Einstein's problem (a logic puzzle with five people, each with five separate attributes), the Eight Queens problem (place eight queens on a chess board so that none can capture each other), and even Sudoku.



The 2005 LRM requires a `foreach` constraint to only have a simple array name, not hierarchical reference. Thus you cannot use a `foreach` constraint in one class to constrain an array in subclass.

6.13.5 Generating an Array of Unique Values

How can you create an array of random unique values? If you try to make a `randc` array, each array element will be randomized independently, and so you are almost certain to get repeated values.

You may be tempted to use a constraint solver to compare every element with every other with nested `foreach`-loops as shown in Sample 6.58. This creates over 4,000 individual constraints, which could slow down simulation.

Sample 6.58 Creating unique array values with `foreach`

```
class UniqueSlow;
  rand bit [7:0] ua[64];
  constraint c {
    foreach (ua[i])         // For every element
      foreach (ua[j])
        if (i != j)         // except the diagonals
          ua[i] != ua[j];  // compare to other elements
  }
endclass
```

Instead, you should use procedural code with a helper class containing a `randc` variable so that you can randomize the same variable over and over.

Sample 6.59 Creating unique array values with a randc helper class

```
class randc8;
  randc bit [7:0] val;
endclass

class LittleUniqueArray;
  bit [7:0] ua [64];     // Array of unique values

  function void pre_randomize;
    randc8 rc8;
    rc8 = new();
    foreach (ua[i]) begin
      assert(rc8.randomize());
      ua[i] = rc8.val;
    end
  endfunction
endclass
```

Next is a more general solution. For example, you may need to assign ID numbers to *N* bus drivers, which are in the range of 0 to MAX-1 where MAX >=*N*.

Sample 6.60 Unique value generator

```
// Create unique random values in a range 0:max-1
class RandcRange;
  randc bit [15:0] value;
  int max_value;  // Maximum possible value

  function new(int max_value = 10);
    this.max_value = max_value;
  endfunction

  constraint c_max_value {value < max_value;}
endclass
```

Sample 6.61 Class to generate a random array of unique values

```
class UniqueArray;
  int max_array_size, max_value;
  rand bit [7:0] a[];       // Array of unique values
  constraint c_size {a.size() inside {[1:max_array_size]};}

  function new(int max_array_size=2, max_value=2);
    this.max_array_size = max_array_size;
    // If max_value is smaller than array size,
    // array could have duplicates, so adjust max_value
    if (max_value < max_array_size)
      this.max_value = max_array_size;
    else
      this.max_value = max_value;
  endfunction

  // Array a[] allocated in randomize(), fill w/unique vals
  function void post_randomize;
    RandcRange rr;
    rr = new(max_value);
    foreach (a[i]) begin
      assert (rr.randomize());
      a[i] = rr.value;
    end
  endfunction

  function void display();
    $write("Size: %3d:", a.size());
    foreach (a[i]) $write("%4d", a[i]);
    $display;
  endfunction
endclass
```

Here is a program that uses the `UniqueArray` class.

Sample 6.62 Using the `UniqueArray` class

```
program automatic test;
  UniqueArray ua;
  initial begin
    ua = new(50);              // Array size = 50

    repeat (10) begin
      assert(ua.randomize());  // Create random array
       ua.display();             // Display values
      end
  end
endprogram
```

6.13.6 Randomizing an Array of Handles

If you need to create multiple random objects, you might create a random array of handles. Unlike an array of integers, you need to allocate all the elements before randomization as the random solver never constructs objects. If you have a dynamic array, allocate the maximum number of elements you may need, and then use a constraint to resize the array. A dynamic array of handles can remain the same size or shrink during randomization, but it can never increase in size.

Sample 6.63 Constructing elements in a random array

```
parameter MAX_SIZE = 10;

class RandStuff;
  rand int value;
endclass

class RandArray;
  rand RandStuff array[];    // DonÕt forget rand!

  constraint c {array.size() inside {[1:MAX_SIZE]}; }

  function new();
    array = new[MAX_SIZE];   // Allocate maximum size
    foreach (array[i])
      array[i] = new();
  endfunction;
endclass

RandArray ra;
initial begin
  ra = new();                // Construct array and all objects
  assert(ra.randomize());    // Randomize and maybe shrink array
  foreach (ra.array[i])
    $display(ra.array[i].value);
end
```

See Section 5.14.4 for more on arrays of handles.

6.14 Atomic Stimulus Generation vs. Scenario Generation

Up until now, you have seen atomic random transactions. You have learned how to make a single random bus transaction, a single network packet, or a single processor instruction. This is a good start; however, your job is to verify that the design works with real-world stimuli. A bus may have long sequences of transactions such as DMA transfers or cache fills. Network traffic consists of extended sequences of packets as

you simultaneously read e-mail, browse a web page, and download music from the net, all in parallel. Processors have deep pipelines that are filled with the code for routine calls, `for`-loops, and interrupt handlers. Generating transactions one at a time is unlikely to mimic any of these scenarios.

6.14.1 An Atomic Generator with History

The easiest way to create a stream of related transactions is to have an atomic generator base some of its random values on ones from previous transactions. The class might constrain a bus transaction to repeat the previous command, such as a write, 80% of the time, and also use the previous destination address plus an increment. You can use the `post_randomize` function to make a copy of the generated transaction for use by the next call to `randomize()`.

This scheme works well for smaller cases but gets into trouble when you need information about the entire sequence ahead of time. For example, the DUT may need to know the length of a sequence of network transactions before it starts.

6.14.2 Randsequence

The next way to generate a sequence of transactions is by using the randsequence construct in SystemVerilog. With `randsequence`, you describe the grammar of the transaction, using a syntax similar to BNF (Backus-Naur Form).

Sample 6.64 Command generator using `randsequence`

```
initial begin
  for (int i=0; i<15; i++) begin
    randsequence (stream)
      stream :  cfg_read := 1 |
                io_read  := 2 |
                mem_read := 5;
      cfg_read : { cfg_read_task; } |
                 { cfg_read_task; } cfg_read;
      mem_read : { mem_read_task; } |
                 { mem_read_task; } mem_read;
      io_read  : { io_read_task; } |
                 { io_read_task; } io_read;
    endsequence
  end // for
end

task cfg_read_task;
  ...
endtask
```

Sample 6.64 generates a sequence called `stream`. A `stream` can be either `cfg_read`, `io_read`, or `mem_read`. The random sequence engine randomly picks one. The `cfg_read` label has a weight of 1, `io_read` has twice the weight and so is twice as likely to be chosen as `cfg_read`. The label `mem_read` is most likely to be chosen, with a weight of 5.

A `cfg_read` can be either a single call to the `cfg_read_task`, or a call to the task followed by another `cfg_read`. As a result, the task is always called at least once, and possibly many times.

One big advantage of `randsequence` is that it is procedural code and you can debug it by stepping though the execution, or adding `$display` statements. When you call `randomize()` for an object, it either all works or all fails, but you can't see the steps taken to get to a result.

There are several problems with using `randsequence`. The code to generate the sequence is separate and a very different style from the classes with data and constraints used by the sequence. So if you use both `randomize()` and `randsequence`, you have to master two different forms of randomization. More seriously, if you want to modify a sequence, perhaps to add a new branch or action, you have to modify the original sequence code. You can't just make an extension. As you will see in Chap. 8, you can extend a class to add new code, data, and constraints without having to edit the original class.

6.14.3 Random Array of Objects

The last form of generating random sequences is to randomize an entire array of objects. You can create constraints that refer to the previous and next objects in the array, and the SystemVerilog solver solves all constraints simultaneously. Since the entire sequence is generated at once, you can then extract information such as the total number of transactions or a checksum of all data values before the first transaction is sent. Alternatively, you can build a sequence for a DMA transfer that is constrained to be exactly 1,024 bytes, and let the solver pick the right number of transactions to reach that goal.

6.14.4 Combining Sequences

You can combine multiple sequences together to make a more realistic flow of transactions. For example, for a network device, you could make one sequence that resembles downloading e-mail, a second that is viewing a web page, and a third that is entering single characters into web-based form.The techniques to combine these flows is beyond the scope of this book, but you can learn more from the VMM, as described in Bergeron et al. (2005).

6.15 Random Control

At this point you may be thinking that this process is a great way to create long streams of random input into your design. Or you may think that this is a lot of work if all you want to do is occasionally to make a random decision in your code. You may prefer a set of procedural statements that you can step through using a debugger.

6.15.1 Introduction to `randcase`

You can use `randcase` to make a weighted choice between several actions, without having to create a class and instance. Sample 6.65 chooses one of the three branches based on the weight. SystemVerilog adds up the weights (1+8+1 = 10), chooses a value in this range, and then picks the appropriate branch. The branches are not order dependent, the weights can be variables, and they do not have to add up to 100%.

Sample 6.65 Random control with `randcase` and `$urandom_range`

```
initial begin
  int len;
  randcase
    1: len = $urandom_range(0, 2);  // 10%: 0, 1, or 2
    8: len = $urandom_range(3, 5);  // 80%: 3, 4, or 5
    1: len = $urandom_range(6, 7);  // 10%: 6 or 7
  endcase
  $display("len=%0d", len);
  end
```

The `$urandom_range` function returns a random number in the specified range. You can specify the arguments as (low, high) or (high, low). If you use just a single argument, SystemVerilog treats it as (0, high).

You can write Sample 6.65 using a class and the `randomize()` function. For this small case, the OOP version is a little larger. However, if this were part of a larger class, the constraint would be more compact than the equivalent `randcase` statement.

Sample 6.66 Equivalent constrained class

```
class LenDist;
  rand int len;
  constraint c
    {len dist {[0:2] := 1, [3:5] := 8, [6:7] := 1}; }
endclass

LenDist lenD;

initial begin
  lenD = new();
  assert (lenD.randomize());
  $display("Chose len=%0d", lenD.len);
  end
```

Code using `randcase` is more difficult to override and modify than random constraints. The only way to modify the random results is to rewrite the code or use variable weights.

Be careful using `randcase`, as it does not leave any tracks behind. For example, you could use it to decide whether or not to inject an error in a transaction. The problem is that the downstream transactors and scoreboard need to know of this choice. The best way to inform them would be to use a variable in the transaction or environment. However, if you are going to create a variable that is part of these classes, you could have made it a random variable and used constraints to change its behavior in different tests.

6.15.2 Building a Decision Tree with `randcase`

You can use `randcase` when you need to create a decision tree. Sample 6.67 has just two levels of procedural code, but you can see how it can be extended to use more.

Sample 6.67 Creating a decision tree with `randcase`

```
initial begin
  // Level 1
  randcase
    one_write_wt: do_one_write();
    one_read_wt:  do_one_read();
    seq_write_wt: do_seq_write();
    seq_read_wt:  do_seq_read();
  endcase
  end

// Level 2
task do_one_write;
  randcase
    mem_write_wt: do_mem_write();
    io_write_wt:  do_io_write();
    cfg_write_wt: do_cfg_write();
  endcase
endtask

task do_one_read;
  randcase
    mem_read_wt: do_mem_read();
    io_read_wt:  do_io_read();
    cfg_read_wt: do_cfg_read();
  endcase
endtask
```

6.16 Random Number Generators

How random is SystemVerilog? On the one hand, your testbench depends on an uncorrelated stream of random values to create stimulus patterns that go beyond any directed test. On the other hand, you need to repeat the patterns over and over during debug of a particular test, even if the design and testbench make minor changes.

6.16.1 Pseudorandom Number Generators

Verilog uses a simple PRNG that you could access with the `$random` function. The generator has an internal state that you can set by providing a seed to `$random`. All IEEE-1364-compliant Verilog simulators use the same algorithm to calculate values.

Sample 6.68 shows a simple PRNG, not the one used by SystemVerilog. The PRNG has a 32-bit state. To calculate the next random value, square the state to produce a 64-bit value, take the middle 32 bits, then add the original value.

Sample 6.68 Simple pseudorandom number generator

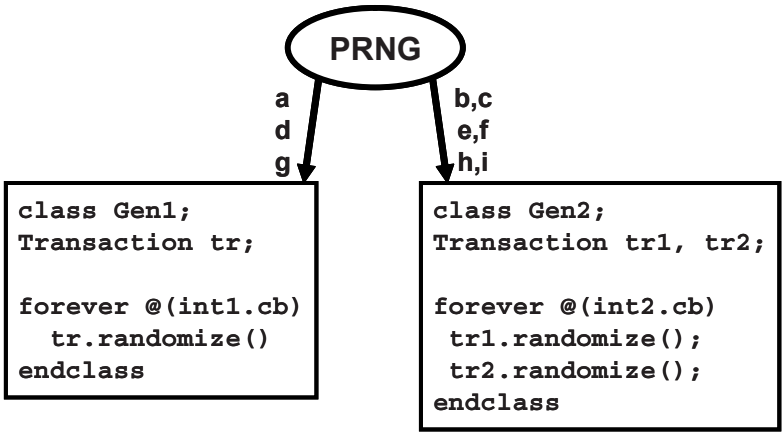
```
reg [31:0] state = 32Õh12345678;
function logic [31:0] my_random;
  logic [63:0] s64;
  s64 = state * state;
  state = (s64 >> 16) + state;
  my_random = state;
endfunction
```

You can see how this simple code produces a stream of values that seem random, but can be repeated by using the same seed value. SystemVerilog calls its PRNG to generate a new value for `randomize()` and `randcase`.

6.16.2 Random Stability – Multiple Generators

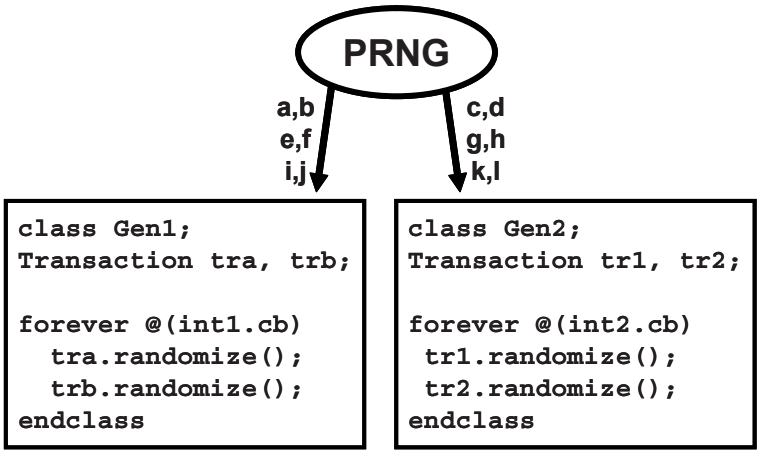
Verilog has a single PRNG that is used for the entire simulation. What would happen if SystemVerilog kept this approach? Testbenches often have several stimulus generators running in parallel, creating data for the design under test. If two streams share the same PRNG, they each get a subset of the random values.

Figure 6-3 Sharing a single random generator



In Figure 6-3, there are two stimulus generators and a single PRNG producing values `a`, `b`, `c`, etc. `Gen2` has two random objects, and so during every cycle, it uses twice as many random values as `Gen1`. A problem can occur when one of the classes changes. `Gen1` gets an additional random variable, and so consumes two random values every time it is called.

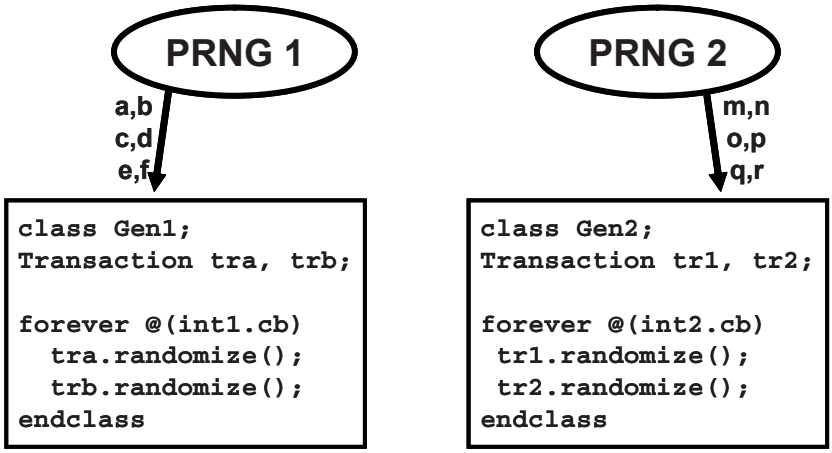
Figure 6-4 First generator uses additional values



This approach changes the values used not only by `Gen1` but also by `Gen2` (Figure 6-4).

In SystemVerilog, there is a separate PRNG for every object and thread. Changes to one object don't affect the random values seen by others (Figure 6-5).

Figure 6-5 Separate random generators per object



6.16.3 Random Stability and Hierarchical Seeding

In SystemVerilog, every object and thread has its own PRNG and unique seed. When a new object or thread is started, its PRNG is seeded from its parent's PRNG. Thus a single seed specified at the start of simulation can create many streams of random stimulus, each distinct.

When you are debugging a testbench, you add, delete, and move code. Even with random stability, your changes may cause the testbench to generate different random values. This can be very frustrating if you are in the middle of debugging a DUT failure, and the testbench no longer creates the same stimulus. You can minimize the effect of code modifications by adding any new objects or threads after existing ones. Sample 6.69 shows a routine from testbench that constructs objects, and runs them in parallel threads.

Sample 6.69 Test code before modification

```
function void build();
  pci_gen gen0, gen1;
  gen0 = new();
  gen1 = new();
  fork
    gen0.run();
    gen1.run();
  join
endfunction : build
```

Sample 6.70 adds a new generator, and runs it in a new thread. The new object is constructed after the existing ones, and the new thread is spawned after the old ones.

Sample 6.70 Test code after modification

```
function void build();
  pci_gen gen0, gen1;
  atm_gen agen;          // New ATM generator
  gen0 = new();
  gen1 = new();
  new_gen = new();       // Construct new object after old ones

  fork
    gen0.run();
    gen1.run();
    new_gen.run();       // Spawn new thread after old ones
  join
endfunction : build
```

Of course as new code is added, you may not be able to keep the random streams the same as the old ones, but you might be able to postpone the undesirable side effects from these changes.

6.17 Random Device Configuration



An important part of your DUT to test is the configuration of both the internal DUT settings and the system that surrounds it. As described in Section 6.2.1, your tests should randomize the environment so that you can be confident it has been tested in as many modes as possible.

Sample 6.71 shows how to create a random testbench configuration and modify its results as needed at the test level. The `eth_cfg` class describes the configuration for a 4-port Ethernet switch. It is instantiated in an environment class, which in turn is used in the test. The test overrides one of the configuration values, enabling all 4 ports.

Sample 6.71 Ethernet switch configuration class

```
class eth_cfg;
  rand bit [ 3:0] in_use;        // Ports used in test
  rand bit [47:0] mac_addr[4];   // MAC addresses
  rand bit [ 3:0] is_100;        // 100mb mode
  rand uint run_for_n_frames;    // # frames in test

  // Force some addr bits when running in unicast mode
  constraint local_unicast {
    foreach (mac_addr[i])
      mac_addr[i][41:40] == 2'b00;
  }

  constraint reasonable {        // Limit test length
    run_for_n_frames inside {[1:100]};
  }
endclass : eth_cfg
```

The configuration class is used in the `Environment` class during several phases. The configuration is constructed in the `Environment` constructor, but not randomized until the `gen_cfg` phase. This allows you to turn constraints on and off before `randomize()` is called. Afterwards, you can override the generated values before the `build` phase creates the virtual components around the DUT.

Sample 6.72 Building environment with random configuration

```
class Environment;
  eth_cfg cfg;
  eth_src gen[4];
  eth_mii drv[4];

  function new();
    cfg = new();              // Construct the cfg
  endfunction

  function void gen_cfg;
    assert(cfg.randomize());  // Randomize the cfg
  endfunction

  // Use random configuration to build the environment
  function void build();
    foreach (gen[i])
      if (cfg.in_use[i]) begin
        gen[i] = new();
        drv[i] = new();
        if (cfg.is_100[i])
           drv[i].set_speed(100);
        end
    endfunction

  task run();
    foreach (gen[i])
      if (cfg.in_use[i]) begin
        // Start the testbench transactors
        gen[i].run();
        ...
    end
  endtask

  task wrap_up();
    // Not currently used
  endtask
endclass : Environment
```

The definitions of classes such as `eth_src` and `eth_mii` are not shown.

Now you have all the components to build a test, which is described in a program block. The test instantiates the environment class and then runs each step.

Sample 6.73 Simple test using random configuration

```
program test;
  Environment env;

  initial begin
    env = new();     // Construct environment
    env.gen_cfg;     // Create random configuration
    env.build();     // Build the testbench environment
    env.run();       // Run the test
    env.wrap_up();   // Clean up after test & report
    end
endprogram
```

You may want to override the random configuration, perhaps to reach a corner case. The following test randomizes the configuration class and then enables all the ports.

Sample 6.74 Simple test that overrides random configuration

```
program test;
  Environment env;

  initial begin
    env = new();     // Construct environment
    env.gen_cfg;     // Create random configuration

    // Override random in-use Ð turn all 4 ports on
    env.cfg.in_use = 4Õb1111;

    env.build();     // Build the testbench environment
    env.run();       // Run the test
    env.wrap_up();   // Clean up after test & report
    end
endprogram
```



Notice how in Sample 6.72 all generators were constructed, but only a few were run, depending on the random configuration. If you only constructed the generators that are in use, you would have to surround any reference to `gen[i]` with a test of `in_use[i]`, otherwise your testbench would crash when it tried to refer to the nonexistent generator. The extra memory taken up by these generators that are not used is a small price to pay for a more stable testbench.

6.18 Conclusion

CRTs are the only practical way to generate the stimulus needed to verify a complex design. SystemVerilog offers many ways to create a random stimulus and this chapter presents many of the alternatives.

A test needs to be flexible, allowing you either to use the values generated by default or to constrain or override the values so that you can reach your goals. Always plan ahead when creating your testbench by leaving sufficient “hooks” so that you can steer the testbench from the test without modifying existing code.

Chapter 7

Threads and Interprocess Communication

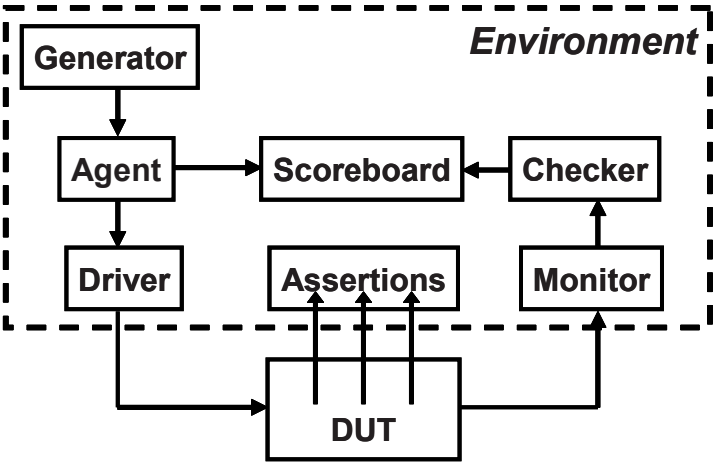
In real hardware, the sequential logic is activated on clock edges, whereas combinational logic is constantly changing when any inputs change. All this parallel activity is simulated in Verilog RTL using `initial` and `always` blocks, plus the occasional gate and continuous assignment statement. To stimulate and check these blocks, your testbench uses many threads of execution, all running in parallel. Most blocks in your testbench environment are modeled with a transactor and run in their own thread.

The SystemVerilog scheduler is the traffic cop that chooses which thread runs next. You can use the techniques in this chapter to control the threads and thus your testbench.

Each of these threads communicates with its neighbors. In Figure 7-1, the generator passes the stimulus to the agent. The environment class needs to know when the generator completes and then tell the rest of the testbench threads to terminate. This is done with interprocess communication (IPC) constructs such as the standard Verilog events, event control and `wait` statements, and the SystemVerilog mailboxes and semaphores.[1]

[1]The SystemVerilog LRM uses "thread" and "process" interchangeably. The term "process" is most commonly associated with Unix processes, in which each contains a program running in its own memory space. Threads are lightweight processes that may share common code and memory, and consume far fewer resources than a typical process. This book uses the term "thread." However, "interprocess communication" is such a common term that it is used in this book.

Figure 7-1 Testbench environment blocks



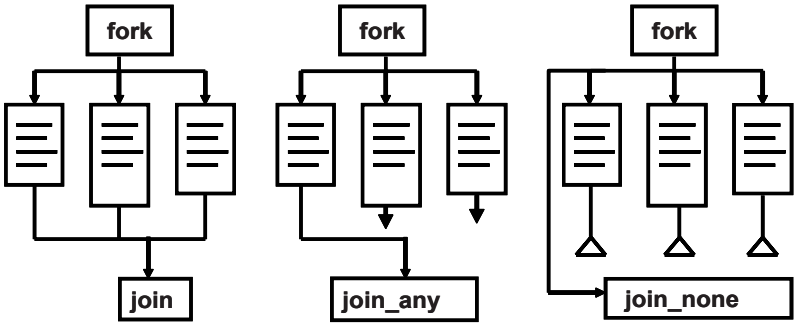
7.1 Working with Threads

While all the thread constructs can be used in both modules and program blocks, your testbenches belong to program blocks. As a result, your code always starts with `initial` blocks that start executing at time 0. You cannot put an `always` block in a program. However, you can easily get around this by using a `forever` loop in an `initial` block.

Classic Verilog has two ways of grouping statements – with a `begin...end` or `fork...join`. Statements in a `begin...end` run sequentially, whereas those in a `fork...join` execute in parallel. The latter is very limited in that all statements inside the `fork...join` have to finish before the rest of the block can continue. As a result, it is rare for Verilog testbenches to use this feature.

SystemVerilog introduces two new ways to create threads – with the `fork...join_none` and `fork...join_any` statements, shown in Figure 7-2.

Figure 7-2 `Fork...join` blocks



Your testbench communicates, synchronizes, and controls these threads with existing constructs such as events, `@` event control, the `wait` and `disable` statements, plus new language elements such as semaphores and mailboxes.

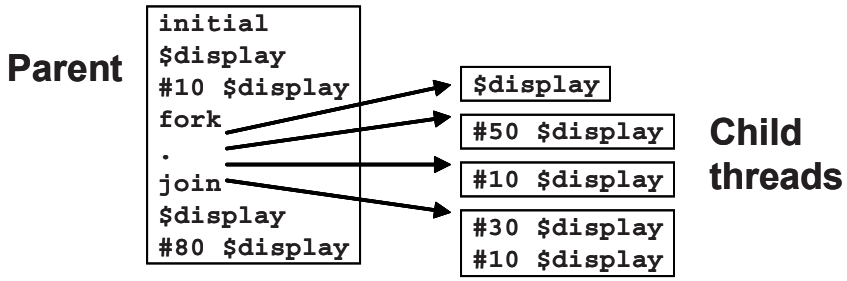
7.1.1 Using `fork...join` and `begin...end`

Sample 7.1 has a `fork...join` parallel block with an enclosed `begin...end` sequential block, and shows the difference between the two.

Sample 7.1 Interaction of `begin...end` and `fork...join`

```
initial begin
      $display("@%0t: start fork...join example", $time);
  #10 $display("@%0t: sequential after #10", $time);
  fork
       $display("@%0t: parallel start", $time);
    #50 $display("@%0t: parallel after #50", $time);
    #10 $display("@%0t: parallel after #10", $time);
    begin
      #30 $display("@%0t: sequential after #30", $time);
      #10 $display("@%0t: sequential after #10", $time);
    end
  join
  $display("@%0t: after join", $time);
  #80 $display("@%0t: finish after #80", $time);
end
```

Figure 7-3 `Fork...join` block



Note in the output below that code in the `fork...join` executes in parallel, and so statements with shorter delays execute before those with longer delays. As shown in Sample 7.2, the `fork...join` completes after the last statement, which starts with `#50`.

Sample 7.2 Output from `begin...end` and `fork...join`

```
@0: start fork...join example
@10: sequential after #10
@10: parallel start
@20: parallel after #10
@40: sequential after #30
@50: sequential after #10
@60: parallel after #50
@60: after join
@140: finish after #80
```

7.1.2 Spawning Threads with `fork...join_none`

A `fork...join_none` block schedules each statement in the block, but execution continues in the parent thread. Sample 7.3 is identical to Sample 7.1 except that the `join` has been converted to `join_none`.

Sample 7.3 `Fork...join_none` code

```
initial begin
  $display("@%0t: start fork...join_none example", $time);
  #10 $display("@%0t: sequential after #10", $time);
  fork
        $display("@%0t: parallel start", $time);
    #50 $display("@%0t: parallel after #50", $time);
    #10 $display("@%0t: parallel after #10", $time);
    begin
      #30 $display("@%0t: sequential after #30", $time);
      #10 $display("@%0t: sequential after #10", $time);
    end
  join_none
  $display("@%0t: after join_none", $time);
  #80 $display("@%0t: finish after #80", $time);
end
```

The diagram for this block is similar to Figure 7-3. Note that the statement after the `join_none` block executes before any statement inside the `fork...join_none`.

Sample 7.4 `Fork...join_none` output

```
@0: start fork...join_none example
@10: sequential after #10
@10: after join_none
@10: parallel start
@20: parallel after #10
@40: sequential after #30
@50: sequential after #10
@60: parallel after #50
@90: finish after #80
```

7.1.3 Synchronizing Threads with `fork...join_any`

A `fork...join_any` block schedules each statement in the block. Then, when the first statement completes, execution continues in the parent thread. All other remaining threads continue. Sample 7.5 is identical to the previous examples, except that the `join` has been converted to `join_any`.

Sample 7.5 `Fork...join_any` code

```
initial begin
  $display("@%0t: start fork...join_any example", $time);
  #10 $display("@%0t: sequential after #10", $time);
  fork
        $display("@%0t: parallel start", $time);
    #50 $display("@%0t: parallel after #50", $time);
    #10 $display("@%0t: parallel after #10", $time);
    begin
      #30 $display("@%0t: sequential after #30", $time);
      #10 $display("@%0t: sequential after #10", $time);
    end
  join_any
  $display("@%0t: after join_any", $time);
  #80 $display("@%0t: finish after #80", $time);
end
```

Note in Sample 7.6, the statement `$display("after join_any")` completes after the first statement in the parallel block.

Sample 7.6 Output from `fork...join_any`

```
@0: start fork...join_any example
@10: sequential after #10
@10: parallel start
@10: after join_any
@20: parallel after #10
@40: sequential after #30
@50: sequential after #10
@60: parallel after #50
@90: finish after #80
```

7.1.4 Creating Threads in a Class

You can use a `fork...join_none` to start a thread, such as the code for a random transactor generator. Sample 7.7 shows a generator/driver class with a `run` task that creates *N* packets. The full testbench has classes for the driver, monitor, checker, and more, all with transactors that need to run in parallel.

Sample 7.7 Generator / Driver class with a run task

```
class Gen_drive;

  // Transactor that creates N packets
  task run(int n);
    Packet p;

    fork
      repeat (n) begin
        p = new();
        assert(p.randomize());
        transmit(p);
      end
    join_none       // Use fork-join_none so run() does not block
  endtask

  task transmit(input Packet p);
  ...
  endtask
endclass

Gen_drive gen;

initial begin
  gen = new();
  gen.run(10);
  // Start the checker, monitor, and other threads
  ...
end
```



There are several points you should notice with Sample 7.7. First, the transactor is not started in the `new()` function. The constructor should just initialize values, not start any threads. Separating the constructor from the code that does the real work allows you to change any variables before you start executing the code in the object. This allows you to inject errors, modify the defaults, and alter the behavior of the object.

Next, the `run` task starts a thread in a `fork...join_none` block. The thread is an implementation detail of the transactor and should be spawned there, not in the parent class.

7.1.5 Dynamic Threads

Verilog's threads are very predictable. You can read the source code and count the `initial`, `always`, and `fork...join` blocks to know how many threads were in a

module. SystemVerilog lets you create threads dynamically, and does not require you to wait for them to finish.

In Sample 7.8, the testbench generates random transactions and sends them to a DUT that stores them for some predetermined time, and then returns them. The testbench has to wait for the transaction to complete, but does not want to stop the generator.

Sample 7.8 Dynamic thread creation

```
program automatic test(bus_ifc.TB bus);
  // Code for interface not shown
  task check_trans(Transaction tr);
    fork
      begin
      wait (bus.cb.addr == tr.addr);
      $display("@%0t: Addr match %d", $time, tr.addr);
      end
    join_none
  endtask

  Transaction tr;

  initial begin
    repeat (10) begin
      // Create a random transaction
      tr = new();
      assert(tr.randomize());

      // Send transaction into the DUT
      transmit(tr);  // Task not shown

      // Wait for reply from DUT
      check_trans(tr);
    end
    #100; // Wait for final transaction to complete
  end
endprogram
```

When the `check_trans` task is called, it spawns off a thread to watch the bus for the matching transaction address. During a normal simulation, many of these threads run concurrently. In this simple example, the thread just prints a message, but you could add more elaborate controls.

7.1.6 Automatic Variables in Threads



A common but subtle bug occurs when you have a loop that spawns threads and you don't save variable values before the next iteration. Sample 7.8 only works in a `program` or `module` with automatic

storage. If `check_trans` used static storage, each thread would share the same variable `tr`, and so later calls would overwrite the value set by earlier ones. Likewise, if the example had the `fork...join_none` inside the `repeat` loop, it would try to match incoming transactions using `tr`, but its value would change the next time through the loop. Always use automatic variables to hold values in concurrent threads.

Sample 7.9 has a `fork...join_none` inside a `for`-loop. SystemVerilog schedules the threads inside a `fork...join_none`, but they are not executed until after the original code blocks, here because of the `#0` delay. So Sample 7.9 prints "`3 3 3`," which are the values of the index variable `j` when the loop terminates.

Sample 7.9 Bad `fork...join_none` inside a loop

```
program no_auto;
  initial begin
    for (int j=0; j<3; j++)
      fork
        $write(j);  // Bug Ð prints final value of index
      join_none
    #0 $display("\n");
  end
endprogram
```

Sample 7.10 Execution of bad `fork...join_none` inside a loop

```
j   Statement
0   for (j=0; ...
0   Spawn $write(j) [thread 0]
1   j++             j=1
1   Spawn $write(j) [thread 1]
2   j++             j=2
2   Spawn $write(j) [thread 2]
3   j++             j=3
3   join_none
3   #0
3   $write(j)       [thread 0: j=3]
3   $write(j)       [thread 1: j=3]
3   $write(j)       [thread 2: j=3]
3   $display(Ô\nÓ)
```

The `#0` delay blocks the current thread and reschedules it to start later during the current time slot. In Sample 7.10, the delay makes the current thread run after the threads spawned in the `fork...join_none` statement. This delay is useful for blocking a thread, but you should be careful, as excessive use causes race conditions and unexpected results.

You should use `automatic` variables inside a `fork...join` statement to save a copy of a variable, as shown in Sample 7.11.

Sample 7.11 Automatic variables in a `fork...join_none`

```
initial begin
  for (int j=0; j<3; j++)
    fork
      automatic int k = j;     // Make copy of index
      $write(k);               // Print copy
    join_none
  #0 $display;
end
```

The `fork...join_none` block is split into two parts. The `automatic` variable declaration with initialization runs in the thread inside the `for`-loop. During each loop, a copy of `k` is created and set to the current value of `j`. Then the body of the `fork...join_none` (`$write`) is scheduled, including a copy of `k`. After the loop finishes, `#0` blocks the current thread, and so the three threads run, printing the value of their copy of `k`. When the threads complete, and there is nothing else left during the current time-slot region, SystemVerilog advances to the next statement and the `$display` executes.

Sample 7.12 traces the code and variables from Sample 7.11. The three copies of the automatic variable `k` are called `k0`, `k1`, and `k2`.

Sample 7.12 Steps in executing automatic variable code

```
j k0 k1 k2  Statement
0           for (j=0; ...
0  0        Create k0, spawn $write(k) [thread 0]
1  0        j++
1  0  1     Create k1, spawn $write(k) [thread 1]
2  0  1     j++
2  0  1  2  Create k2, spawn $write(k) [thread 2]
3  0  1  2  j<3
3  0  1  2  join_none
3  0  1  2  #0
3  0  1  2  $write(k0)        [thread 0]
3  0  1  2  $write(k1)        [thread 1]
3  0  1  2  $write(k2)        [thread 2]
3  0  1  2  $display(Õ\nŐ)
```

The good news is that you do not have to use the `automatic` keyword in the declaration if this code is in a program or module that uses automatic storage. If you are using the guideline in Section 3.6.1, you are already covered. All you need to remember is to make a copy of the loop variable.

Another way to write Sample 7.11 is to declare the automatic variable outside of the `fork...join_none`. Sample 7.13 works inside a program with automatic storage.

Sample 7.13 Automatic variables in a `fork...join_none`

```
program automatic bug_free;
  initial begin
    for (int j=0; j<3; j++) begin
      int k = j;                // Make copy of index
      fork
        $write(k);              // Print copy
      join_none
    end
    #0 $display;
  end
endprogram
```

7.1.7 Waiting for all Spawned Threads

In SystemVerilog, when all the `initial` blocks in the program are done, the simulator exits. Sample 7.14 shows how you can spawn many threads, which might still be running. Use the `wait fork` statement to wait for all child threads.

Sample 7.14 Using `wait fork` to wait for child threads

```
task run_threads;
  ...                   // Create some transactions
  fork
    check_trans(tr1);   // Spawn first thread
    check_trans(tr2);   // Spawn second thread
    check_trans(tr3);   // Spawn third thread
  join_none
  ...                   // Do some other work

  // Now wait for the above threads to complete
  wait fork;
endtask
```

7.1.8 Sharing Variables Across Threads



Inside a class's routines, you can use local variables, class variables, or variables defined in the program. If you forget to declare a variable, SystemVerilog looks up the higher scopes until it finds a match. This can cause subtle bugs if two parts of the code are unintentionally sharing the same variable, perhaps because you forgot to declare it in the innermost scope.

For example, if you like to use the index variable, `i`, be careful that two different threads of your testbench don't concurrently modify this variable by each using it in a

`for`-loop. Or you may forget to declare a local variable in a class, such as `Buggy`, shown below. If your program block declares a global `i`, the class just uses the global instead of the local that you intended. You might not even notice this unless two parts of the program try to modify the shared variable at the same time.

Sample 7.15 Bug using shared program variable

```
program bug;

  class Buggy;
    int data[10];
    task transmit;
      fork
        for (i=0; i<10; i++)  // i is not declared here
          send(data[i]);
      join_none
    endtask
  endclass

  int i;                      // Program-level i, shared
  Buggy b;
  event receive;

  initial begin
    b = new();
    for (i=0; i<10; i++)      // i is not declared here
      b.data[i] = i;
    b.transmit();

    for (i=0; i<10; i++)      // i is not declared here
      @(receive) $display(b.data[i]);
    end
endprogram
```

The solution is to declare all your variables in the smallest scope that encloses all uses of the variable. In Sample 7.15, declare the index variables inside the `for`-loops, not at the program or scope level. Better yet, use the `foreach` statement whenever possible.

7.2 Disabling Threads

Just as you need to create threads in the testbench, you also need to stop them. The Verilog `disable` statement works on SystemVerilog threads.

7.2.1 Disabling a Single Thread

Here is the `check_trans` task, this time using a `fork...join_any` plus a `disable` to create a watch with a time-out. In this case, you are disabling a label to precisely specify the block to stop.

The task and outermost `fork...join_none` are identical to Sample 7.8. This version has two threads inside a `fork...join_any` such that the simple `wait` statement is done in parallel with a delayed display. If the correct bus address comes back quickly enough, the `wait` construct completes, the `join_any` executes, and then the `disable` kills off the remaining thread. However, if the bus address does not get the right value before the `TIME_OUT` delay completes, the error message is printed, the `join_any` executes, and the `disable` kills the thread with the `wait`.

Sample 7.16 Disabling a thread

```
parameter TIME_OUT = 1000;

task check_trans(Transaction tr);
  fork

    begin
      // Wait for response, or some maximum delay
      fork : timeout_block
        begin
          wait (bus.cb.addr == tr.addr);
          $display("@%0t: Addr match %d", $time, tr.addr);
        end
        #TIME_OUT $display("@%0t: Error: timeout", $time);
      join_any
      disable timeout_block;
    end

  join_none
endtask
```

7.2.2 Disabling Multiple Threads

Sample 7.16 used the classic Verilog `disable` statement to stop the threads in a named block. SystemVerilog introduces the `disable fork` statement so that you can stop all child threads that have been spawned from the current thread.



Watch out, as you might unintentionally stop too many threads, such as those created from task calls. You should always surround the target code with a `fork...join` to limit the scope of a `disable fork` statement. Sample 7.17 has an additional `begin...end` block inside the `fork...join` to make the statements sequential.

The following sections show how you can asynchronously disable multiple threads. This can cause unexpected behavior, and so you should watch out for side effects when a thread is stopped midstream. You may instead want to design your algorithm to check for interrupts at stable points, and then gracefully give up its resources.

The next few examples use the `check_trans` task from Sample 7.16. You can just think of this task as doing a `#TIME_OUT`.

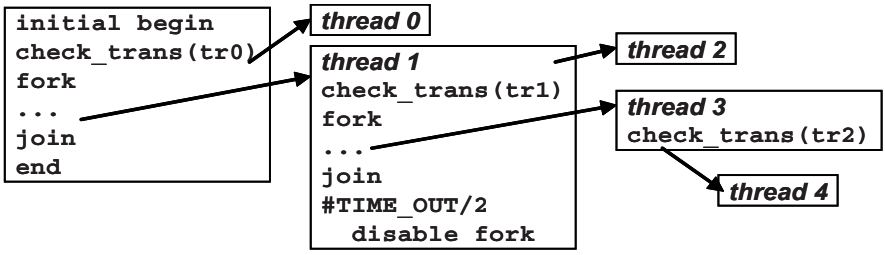
Sample 7.17 Limiting the scope of a `disable fork`

```
initial begin
  check_trans(tr0);       // Thread 0

  // Create a thread to limit scope of disable
  fork                    // Thread 1
    begin
      check_trans(tr1);     // Thread 2
      fork                  // Thread 3
        check_trans(tr2);   // Thread 4
      join

      // Stop threads 1-4, but leave 0 alone
      #(TIME_OUT/2) disable fork;
   end
  join
end
```

Figure 7-4 `Fork...join` block diagram



The code calls `check_trans` that starts thread 0. Next a `fork...join` creates thread 1. Inside this thread, one is spawned by the `check_trans` task and one by the innermost `fork...join`, which spawns thread 4 by calling the task (Figure 7-4). After a delay, a `disable fork` stops thread 1 and all its children, 2–4. Thread 0 is outside the `fork...join` block that has the `disable`, and so it is unaffected.

Sample 7.18 is the more robust version of Sample 7.17, with `disable` with a label that explicitly names the threads that you want to stop.

Sample 7.18 Using `disable` label to stop threads

```
initial begin
  check_trans(tr0);      // Thread 0
  fork                   // Thread 1
    begin : threads_inner
      check_trans(tr1);  // Thread 2
      check_trans(tr2);  // Thread 3
    end

    // Stop threads 2 & 3, but leave 0 alone
    #(TIME_OUT/2) disable threads_inner;
  join
end
```

7.2.3 Disable a Task that was Called Multiple Times

Be careful when you disable a block from inside that block – you might end up stopping more than you expected. As expected, if you disable a task from inside the task, it is like a return statement, but it also kills all threads started by the task. If the task has been called from multiple threads, disabling one will disable them all.

In Sample 7.19, the `wait_for_time_out` task is called three times, spawning three threads. In addition, thread 0 also disables the task after #2. When you run this code, you will see the three threads starting, but none finishes, because of the `disable` in thread 0.

Sample 7.19 Using `disable` label to stop a task

```
task wait_for_time_out(int id);
  if (id == 0)
    fork
      begin
        #2;
        $display("@%0t: disable wait_for_time_out", $time);
        disable wait_for_time_out;
      end
    join_none

  fork : just_a_little
    begin
      $display("@%0t: %m: %0d entering thread", $time, id);
      #TIME_OUT;
      $display("@%0t: %m: %0d done", $time, id);
    end
  join_none
endtask

initial begin
  wait_for_time_out(0); // Spawn thread 0
  wait_for_time_out(1); // Spawn thread 1
  wait_for_time_out(2); // Spawn thread 2
  #(TIME_OUT*2) $display("@%0t: All done", $time);
end
```

7.3 Interprocess Communication

All these threads in your testbench need to synchronize and exchange data. At the most basic level, one thread waits for another, such as the environment object waiting for the generator to complete. Multiple threads might try to access a single resource such as bus in the DUT, and so the testbench needs to ensure that one and only one thread is granted access. At the highest level, threads need to exchange data such as transaction objects that are passed from the generator to the agent. All of this data exchange and control synchronization is called IPC, which is done in SystemVerilog with events, semaphores, and mailboxes. These are described in the remainder of this chapter.

7.4 Events

A Verilog event synchronizes threads. It is similar to a phone, where one person waits for a call from another person. In Verilog a thread waits for an event with the `@` operator. This operator is edge sensitive, and so it always blocks, waiting for the event to change. Another thread triggers the event with the `->` operator, unblocking the first thread.

SystemVerilog enhances the Verilog event in several ways. An event is now a handle to a synchronization object that can be passed around to routines. This feature allows you to share events across objects without having to make the events global. The most common way is to pass the event into the constructor for an object.

There is always the possibility of a race condition in Verilog where one thread blocks on an event at the same time another triggers it. If the triggering thread executes before the blocking thread, the trigger is missed. SystemVerilog introduces the `triggered()` method that lets you check whether an event has been triggered, including during the current time-slot. A thread can wait on this function instead of blocking with the `@` operator.

7.4.1 Blocking on the Edge of an Event

When you run Sample 7.20, one initial block starts, triggers its event, and then blocks on the other event, as shown in the output in Sample 7.21. The second block starts, triggers its event (waking up the first), and then blocks on the first event. However, the second thread locks up because it missed the first event, as it is a zero-width pulse.

Sample 7.20 Blocking on an event in Verilog

```
event e1, e2;
initial begin
  $display("@%0t: 1: before trigger", $time);
  -> e1;
  @e2;
  $display("@%0t: 1: after trigger", $time);
end

initial begin
  $display("@%0t: 2: before trigger", $time);
  -> e2;
  @e1;
  $display("@%0t: 2: after trigger", $time);
 end
```

Sample 7.21 Output from blocking on an event

```
@0: 1: before trigger
@0: 2: before trigger
@0: 1: after trigger
```

7.4.2 Waiting for an Event Trigger

Instead of the edge-sensitive block `@e1`, use the level-sensitive `wait(e1.triggered())`. This does not block if the event has been triggered during this time step. Otherwise, it waits until the event is triggered.

Sample 7.22 Waiting for an event

```
event e1, e2;

initial begin
  $display("@%0t: 1: before trigger", $time);
  -> e1;
  wait (e2.triggered());
  $display("@%0t: 1: after trigger", $time);
end

initial begin
  $display("@%0t: 2: before trigger", $time);
  -> e2;
  wait (e1.triggered());
  $display("@%0t: 2: after trigger", $time);
end
```

When you run Sample 7.22, one initial block starts, triggers its event, and then blocks on the other event. The second block starts, triggers its event (waking up the first) and then blocks on the first event, producing the output in Sample 7.23.

Sample 7.23 Output from waiting for an event

```
@0: 1: before trigger
@0: 2: before trigger
@0: 1: after trigger
@0: 2: after trigger
```

Several of these examples have race conditions and may not execute exactly the same on every simulator. For example, the output in Sample 7.23 assumes that when the second block triggers `e2`, execution jumps back to the first block. It would also be legal for the second block to trigger `e2`, wait on `e1`, and display a message before control is returned back to the first block.

7.4.3 Using Events in a Loop

You can synchronize two threads with an event, but use caution.



If you use `wait(handshake.triggered())` in a loop, be sure to advance the time before waiting again. Otherwise your code will go into a zero delay loop as the `wait` continues over and over again on a single event trigger. Sample 7.24 incorrectly uses a level-sensitive blocking statement for notification that a transaction is ready.

Sample 7.24 Waiting on event causes a zero delay loop

```
forever begin
  // This is a zero delay loop!
  wait(handshake.triggered());
  $display("Received next event");
  process_in_zero_time();
end
```

Just as you learned to always put a delay inside an `always` blocks, you need to put a delay in a transaction process loop. The edge-sensitive delay statement in Sample 7.25 continues once and only once per event trigger.

Sample 7.25 Waiting for an edge on an event

```
forever begin
  // This prevents a zero delay loop!
  @handshake;
  $display(ÒReceived next eventÓ);
  process_in_zero_time();
end
```

You should avoid events if you need to send multiple notifications in a single time slot, and look at other IPC methods with built-in queuing such as semaphores and mailboxes, discussed later in this chapter.

7.4.4 Passing Events

As described above, an event in SystemVerilog can be passed as an argument to a routine. In Sample 7.26, an event is used by a transactor to signal when it has completed.

Sample 7.26 Passing an event into a constructor

```
class Generator;
  event done;
  function new (event done); // Pass event from TB
    this.done = done;
  endfunction

  task run();
    fork
      begin
      ...                    // Create transactions
      -> done;               // Tell the test we are done
      end
    join_none
  endtask
endclass

program automatic test;
  event gen_done;
  Generator gen;

  initial begin
    gen = new(gen_done);         // Instantiate testbench
    gen.run();                   // Run transactor
    wait(gen_done.triggered()); // Wait for finish
  end
endprogram
```

7.4.5 Waiting for Multiple Events

In Sample 7.26, you had a single generator that fired a single event. What if your testbench environment class must wait for multiple child processes to finish, such as *N* generators? The easiest way is to use `wait fork`, which waits for all child processes to end. The problem is that this also waits for all the transactors, drivers, and any other threads that were spawned by the environment. You need to be more selective. You still want to use events to synchronize between the parent and child threads.

You could use a `for`-loop in the parent to wait for each event, but that would only work if thread 0 finished before thread 1, which finished before thread 2, etc. If the threads finish out of order, you could be waiting for an event that triggered many cycles ago.

The solution is to make a new thread and then spawn children from there that each block on an event for each generator, as shown in Sample 7.27. Now you can do a `wait fork` because you are being more selective.

Sample 7.27 Waiting for multiple threads with `wait fork`

```
event done[N_GENERATORS];

initial begin
  foreach (gen[i]) begin
    gen[i] = new();         // Create N generators
    gen[i].run(done[i]);    // Start them running
  end

  // Wait for all gen to finish by waiting for each event
  foreach (gen[i])
    fork
      automatic int k = i;
      wait (done[k].triggered());
    join_none

  wait fork;  // Wait for all those triggers to finish
end
```

Another way to solve this problem is to keep track of the number of events that have triggered, as shown in Sample 7.28.

Sample 7.28 Waiting for multiple threads by counting triggers

```
event done[N_GENERATORS];
int done_count;

initial begin
  foreach (gen[i]) begin
    gen[i] = new();         // Create N generators
    gen[i].run(done[i]);    // Start them running
  end

  // Wait for all generators to finish
  foreach (gen[i])
    fork
      automatic int k = i;
      begin
        wait (done[k].triggered());
        done_count++;
      end
    join_none
  wait (done_count==N_GENERATORS); // Wait for triggers

end
```

That was slightly less complicated. Why not get rid of all the events and just wait on a count of the number of running generators? This count can be a static variable in the

`Generator` class. Note that most of the thread manipulation code has been replaced with a single `wait` construct.

The last block in Sample 7.29 waits for the count using the class scope resolution operator, `::`. You could have used any handle, such as `gen[0]`, but that would be less direct.

Sample 7.29 Waiting for multiple threads using a thread count

```
class Generator;
  static int thread_count = 0;

  task run();
    thread_count++;       // Start another thread
    fork
      begin
      // Do the real work in here
      // And when done, decrement the thread count
      thread_count--;
      end
    join_none
  endtask
endclass

Generator gen[N_GENERATORS];

initial begin
  // Create N generators
  foreach (gen[i])
    gen[i] = new();

  // Start them running
  foreach (gen[i])
    gen[i].run();

  // Wait for all the generators to complete
  wait (Generator::thread_count == 0);
end
```

7.5 Semaphores

A semaphore allows you to control access to a resource. Imagine that you and your spouse share a car. Obviously, only one person can drive it at a time. You can manage this situation by agreeing that whoever has the key can drive it. When you are done with the car, you give up the car so that the other person can use it. The key is the semaphore that makes sure only one person has access to the car. In operating system

terminology, this is known as "mutually exclusive access," and so a semaphore is known as a "mutex" and is used to control access to a resource.

Semaphores can be used in a testbench when you have a resource, such as a bus, that may have multiple requestors from inside the testbench but, as part of the physical design, can only have one driver. In SystemVerilog, a thread that requests a key when one is not available always blocks. Multiple blocking threads are queued in FIFO order.

7.5.1 Semaphore Operations

There are three basic operations for a semaphore. You create a semaphore with one or more keys using the `new` method, get one or more keys with `get`, and return one or more keys with `put`. If you want to try to get a semaphore, but not block, use the `try_get()` function. It returns 1 if there are enough keys, and 0 if there are insufficient keys, as shown in Sample 7.30.

Sample 7.30 Semaphores controlling access to hardware resource

```
program automatic test(bus_ifc.TB bus);
  semaphore sem;          // Create a semaphore
  initial begin
    sem = new(1);         // Allocate with 1 key
    fork
      sequencer();        // Spawn two threads that both
      sequencer();        //   do bus transactions
    join
  end

  task sequencer;
    repeat($urandom%10)   // Random wait, 0-9 cycles
      @bus.cb;
    sendTrans();          // Execute the transaction
  endtask

  task sendTrans;
    sem.get(1);           // Get the key to the bus
    @bus.cb;              // Drive signals onto bus
    bus.cb.addr <= t.addr;
    ...
    sem.put(1);           // Put it back when done
  endtask
endprogram
```

7.5.2 Semaphores with Multiple Keys

There are two things you should watch out for with semaphores. First, you can put more keys back than you took out. Suddenly you may have two keys but only one car! Secondly, be very careful if your testbench needs to get and put multiple keys. Perhaps you have one key left, and a thread requests two, causing it to block. Now a second thread requests a single semaphore – what should happen? In SystemVerilog, the second request, `get(1)`, sneaks ahead of the earlier `get(2)`, bypassing the FIFO ordering.

If you are mixing different-sized requests, you can always write your own class. That way you can be very clear on who gets priority.

7.6 Mailboxes

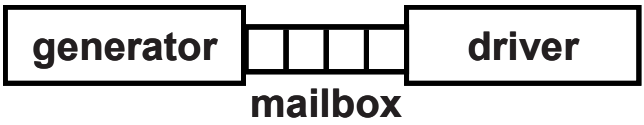
How do you pass information between two threads? Perhaps your generator needs to create many transactions and pass them to a driver. You might be tempted to just have the generator thread call a task in the driver. If you do that, the generator needs to know the hierarchical path to the driver task, making your code less reusable. Additionally, this style forces the generator to run at the same speed as the driver, which can cause synchronization problems if one generator needs to control multiple drivers.



Think of your generator and driver as transactors that are autonomous objects that communicate through a channel. Each object gets a transaction from an upstream object (or creates it, as in the case of a generator), does some processing, and then passes it to a downstream object. The channel must allow its driver and receiver to operate asynchronously. You may be tempted to just use a shared array or queue, but it can be difficult to create code that reads, writes, and blocks between threads.

The solution is a SystemVerilog mailbox. From a hardware point of view, the easiest way to think about a mailbox is that it is just a FIFO, with a source and sink. The source puts data into the mailbox, and the sink gets values from the mailbox. Mailboxes can have a maximum size or can be unlimited. When the source thread tries to put a value into a sized mailbox that is full, it blocks until the value is removed. Likewise, if a sink thread tries to remove a value from a mailbox that is empty, it blocks until a value is put into the mailbox. Figure 7.5 shows a mailbox connecting a generator and driver.

Figure 7-5 A mailbox connecting two transactors



A mailbox is an object and thus has to be instantiated by calling the `new` function. This takes an optional `size` argument to limit the number of entries in the mailbox. If the size is 0 or not specified, the mailbox is unbounded and can hold an unlimited number of entries.

You put data into a mailbox with the `put` task, and remove it with the `get` task. A `put` can block if the mailbox is full and a `get` blocks if it is empty. The `peek` task gets a copy of the data in the mailbox but does not remove it.



The data can be a single value, such as an integer, or logic of any size. You can put a handle into a mailbox, not an object. By default, a mailbox does not have a type, and so you can put any mix of data into it. Don't do it! Stick to one data type per mailbox.

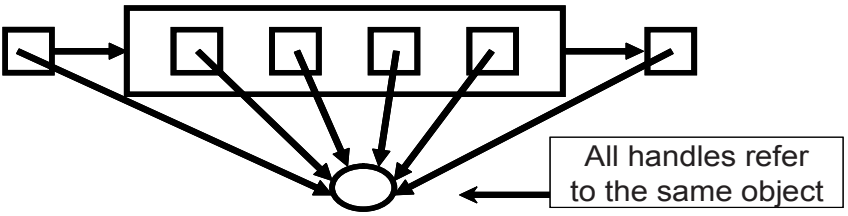


A classic bug, shown in Sample 7.31, is a loop that randomizes objects and puts them in a mailbox, but the object is constructed only once, outside the loop. Since there is only one object, it is randomized over and over. Figure 7-6 shows all the handles pointing to a single object. A mailbox only holds handles, not objects, and so you end up with a mailbox containing multiple handles that all point to the single object. The code that gets the handles from the mailbox just sees the last set of random values.

Sample 7.31 Bad generator creates only one object

```
task generator_bad(int n, mailbox mbx);
  Transaction t;
  t = new();                // Create just one transaction
  repeat (n) begin
    assert(t.randomize());  // Randomize variables
    $display("GEN: Sending addr=%h", t.addr);
    mbx.put(t);             // Send transaction to driver
  end
endtask
```

Figure 7-6 A mailbox with multiple handles to one object



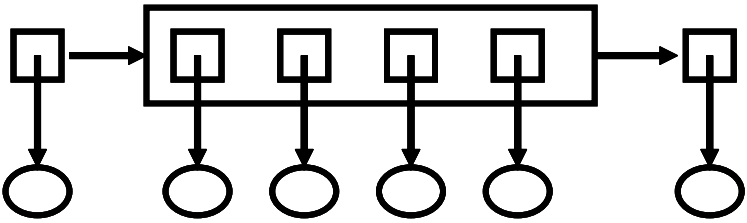
The solution, shown in Sample 7.32, is to make sure your loop has all three steps of constructing the object, randomizing it, and putting it in the mailbox. This bug is so common that it is also mentioned in Section 5.14.3.

Sample 7.32 Good generator creates many objects

```
task generator_good(int n, mailbox mbx);
  Transaction t;
  repeat (n) begin
    t = new();              // Create a new transaction
    assert(t.randomize());  // Randomize variables
    $display("GEN: Sending addr=%h", t.addr);
    mbx.put(t);             // Send transaction to driver
  end
endtask
```

The result, shown in Figure 7-7, is that every handle points to a unique object. This type of generator is known as the Blueprint Pattern and is described in Section 8.2.

Figure 7-7 A mailbox with multiple handles to multiple objects



Sample 7.33 shows the driver that waits for transactions from the generator.

Sample 7.33 Good driver receives transactions from mailbox

```
task driver(mailbox mbx);
  Transaction t;
  forever begin
    mbx.get(t);              // Get transacton from mailbox
    $display("DRV: Received addr=%h", t.addr);
    // Drive transaction into DUT
  end
endtask
```

If you don't want your code to block when accessing the mailbox, use the `try_get()` and `try_peek`() functions. If they are successful, they return a nonzero value; otherwise, they return 0. These are more reliable than the `num` function, as the number of entries can change between when you measure it and when you next access the mailbox.

7.6.1 Mailbox in a Testbench

Sample 7.34–7.36 show a Generator and Driver exchanging transactions using a mailbox, and the top-level program. Note that the two classes need to be defined inside the program block so that they will see the definition of the bus interface.

Sample 7.34 Exchanging objects using a mailbox: the `Generator` class

```
class Generator;
  Transaction tr;
  mailbox mbx;

  function new(mailbox mbx);
    this.mbx = mbx;
  endfunction

  task run(int count);
    repeat (count) begin
      tr = new();
      assert(tr.randomize);
      mbx.put(tr);    // Send out transaction
    end
  endtask

endclass
```

Sample 7.35 shows the matching driver class.

Sample 7.35 Exchanging objects using a mailbox: the `Driver` class

```
class Driver;
  Transaction tr;
  mailbox mbx;

  function new(mailbox mbx);
    this.mbx = mbx;
  endfunction

  task run(int count);
    repeat (count) begin
      mbx.get(tr);      // Fetch next transaction
      @(posedge bus.cb.ack);
      bus.cb.kind <= tr.kind;
      ...
    end
  endtask
endclass
```

Sample 7.36 Exchanging objects using a mailbox: the program block

```
program automatic mailbox_example(bus_if.TB bus, ...);
Ôinclude "transaction.sv"
Ôinclude "generator.sv"
Ôinclude "driver.sv"

  mailbox mbx;              // Mailbox connecting gen & drv
  Generator gen;
  Driver drv;
  int count;

  initial begin
    count = $urandom_range(50);
    mbx = new();            // Construct the mailbox
    gen = new(mbx);         // Construct the generator
    drv = new(mbx);         // Construct the driver
    fork
      gen.run(count);       // Spawn the generator
      drv.run(count);       // Spawn the driver
    join                    // Wait for both to finish
  end
endprogram
```

7.6.2 Bounded Mailboxes

By default, mailboxes are similar to an unlimited FIFO – a producer can put any number of objects into a mailbox before the consumer gets the objects out. However, you may want the two threads to operate in lockstep so that the producer blocks until the consumer is done with the object.

You can specify a maximum size for the mailbox when you construct it. The default mailbox size is 0, which creates an unbounded mailbox. Any size greater than 0 creates a bounded mailbox. If you attempt to put more objects than this limit, `put` blocks until you `get` an object from the mailbox, creating a vacancy.

Sample 7.37 Bounded mailbox

```
Ôtimescale 1ns/1ns
program automatic bounded;
  mailbox mbx;

  initial begin
    mbx = new(1);  // Mailbox size = 1
    fork

      // Producer thread
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
        $display("Producer: after put(%0d)", i);
      end

      // Consumer thread
      repeat(4) begin
        int j;
        #1ns mbx.get(j);
        $display("Consumer: after  get(%0d)", j);
      end

    join
  end
endprogram
```

Sample 7.37 creates the smallest possible mailbox, which can hold a single message. The Producer thread tries to put three messages (integers) in the mailbox, and the Consumer thread slowly gets messages every 1 ns. As Sample 7.38 shows, the first `put` succeeds, and then the Producer tries `put(2)`, which blocks. The Consumer wakes up, gets a message 1 from the mailbox, and so now the Producer can finish putting the message 2.

Sample 7.38 Output from bounded mailbox

```
Producer: before put(1)
Producer: after  put(1)
Producer: before put(2)
Consumer: after  get(1)
Producer: after  put(2)
Producer: before put(3)
Consumer: after  get(2)
Producer: after  put(3)
Consumer: after  get(3)
```

The bounded mailbox acts as a buffer between the two processes. You can see how the Producer generates the next value before the Consumer reads the current value.

7.6.3 Unsynchronized Threads Communicating with a Mailbox



In many cases, two threads that are connected by a mailbox should run in lockstep, so that the producer does not get ahead of the consumer. The benefit of this approach is that your entire chain of stimulus generation now runs in lock step. The highest level generator only completes when the last low level transaction completes transmission. Now your testbench can tell precisely when all stimulus has been sent. In another example, if your generator gets ahead of the driver, and you are gathering functional coverage on the generator, you might record that some transactions were tested, even if the test stopped prematurely. So even though a mailbox allows you to decouple the two sides, you may still want to keep them synchronized.

If you want two threads to run in lockstep, you need a handshake in addition to the mailbox. In Sample 7.39, the Producer and Consumer are now classes that exchange integers using a mailbox, with no explicit synchronization between the two objects. As a result, as shown in Sample 7.40, the producer runs to completion before the consumer even starts.

Sample 7.39 Producer–consumer without synchronization

```
program automatic unsynchronized;

  mailbox mbx;

  class Producer;
    task run();
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
      end
    endtask
  endclass

  class Consumer;
    task run();
      int i;
      repeat (3) begin
        mbx.get(i);      // Get integer from mbx
        $display("Consumer: after  get(%0d)", i);
      end
    endtask
  endclass

  Producer p;
  Consumer c;

initial begin
    // Construct mailbox, producer, consumer
    mbx = new();     // Unbounded
    p = new();
    c = new();

    // Run the producer and consumer in parallel
    fork
      p.run();
      c.run();
    join
 end
endprogram
```

Sample 7.39 has no synchronization and so the Producer puts all three integers into the mailbox before the Consumer can get the first one. This is because a thread continues running until there is a blocking statement, and the Producer has none. The Consumer thread blocks on the first call to `mbx.get`.

Sample 7.40 Producer–consumer without synchronization output

```
Producer: before put(1)
Producer: before put(2)
Producer: before put(3)
Consumer: after  get(1)
Consumer: after  get(2)
Consumer: after  get(3)
```

This example has a race condition, and so on some simulators, the consumer could activate earlier. The result is still the same as the values are determined by the producer, not by how quickly the consumer sees them.

7.6.4 Synchronized Threads Using a Bounded Mailbox and a Peek

In a synchronized testbench, the Producer and Consumer operate in lock step. This way, you can tell when the input stimuli is complete by waiting for any of the threads. If the threads operate unsynchronized, you need to add extra code to detect when the last transaction is applied to the DUT.

To synchronize two threads, the Producer creates and puts a transaction into a mailbox, and then blocks until the Consumer finishes with it. This is done by having the Consumer remove the transaction from the mailbox only when it is finally done with it, not when the transaction is first detected.

Sample 7.41 show the first attempt to synchronize two threads, this time with a bounded mailbox. The Consumer uses the built-in mailbox method `peek()` to look at the data in the mailbox without removing. When the Consumer is done processing the data, it removes the data with `get()`. This frees up the Producer to generate a new value. If the Consumer loop started with a `get()` instead of the `peek()`, the transaction would be immediately removed from the mailbox, and so the Producer could wake up before the Consumer finished with the transaction.

Sample 7.41 Producer–consumer synchronized with bounded mailbox

```
program automatic synch_peek;
// Uses Producer from Sample 7.39

  mailbox mbx;

  class Consumer;
    task run();
      int i;
      repeat (3) begin
        mbx.peek(i);      // Peek integer from mbx
        $display("Consumer: after  get(%0d)", i);
        mbx.get(i);       // Remove from mbx
      end
    endtask
  endclass

  initial begin
    // Construct mailbox, producer, consumer
    mbx = new(1);   // Bounded mailbox - limit 1!
    p = new();
    c = new();

    // Run the producer and consumer in parallel
    fork
      p.run();
      c.run();
    join
 end
endprogram
```

Sample 7.42 Output from producer–consumer with bounded mailbox

```
Producer: before put(1)
Producer: before put(2)
Consumer: after  get(1)
Consumer: after  get(2)
Producer: before put(3)
Consumer: after  get(3)
```

You can see that the Producer and Consumer are in lockstep, but the Producer is still one transaction ahead of the Consumer. This is because a bounded mailbox with size=1 only blocks when you try to do a `put` of the second transaction.[2]

[2]This behavior is different from the VMM channel. If you set a channel's full level to 1, the very first call to put() places the transaction in the channel, but does not return until the transaction is removed.

7.6.5 Synchronized Threads Using a Mailbox and Event

You may want the two threads to use a handshake so that the Producer does not get ahead of the Consumer. The Consumer already blocks, waiting for the Producer using a mailbox. The Producer needs to block, waiting for the Consumer to finish the transaction. This is done by adding a blocking statement to the Producer such as an event, a semaphore, or a second mailbox.

Sample 7.43 uses an event to block the Producer after it puts data in the mailbox. The Consumer triggers the event after it consumes the data.



If you use `wait(handshake.triggered())` in a loop, be sure to advance the time before waiting again, as previously shown in Section 7.4.3. This `wait` blocks only once in a given time slot, and so you need move into another. Sample 7.43 uses the edge-sensitive blocking statement `@handshake` instead to ensure that the Producer stops after sending the transaction. The edge-sensitive statement works multiple times in a time slot but may have ordering problems if the trigger and block happen in the same time slot.

Sample 7.43 Producer–consumer synchronized with an event

```
program automatic mbx_evt;

  event handshake;

  class Producer;
    task run;
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
        @handshake;
        $display("Producer: after put(%0d)", i);
      end
    endtask
  endclass

  // Continued in Sample 7.44
```

Sample 7.44 Producer–consumer synchronized with an event, continued

```
  class Consumer;
    task run;
      int i;
      repeat (3) begin
        mbx.get(i);
        $display("Consumer: after  get(%0d)", i);
        ->handshake;
      end
    endtask
  endclass

  initial begin
    p = new();
    c = new();

    // Run the producer and consumer in parallel
    fork
      p.run();
      c.run();
    join
 end
endprogram
```

Now the Producer does not advance until the Consumer triggers the event, as shown in Sample 7.45.

Sample 7.45 Output from producer–consumer with event

```
Producer: before put(1)
Consumer: after  get(1)
Producer: after  put(1)
Producer: before put(2)
Consumer: after  get(2)
Producer: after  put(2)
Producer: before put(3)
Consumer: after  get(3)
Producer: after  put(3)
```

You can see that the Producer and Consumer are successfully running in lockstep by the fact that the Producer never produces a new value until after the old one is read by the Consumer.

7.6.6 Synchronized Threads Using Two Mailboxes

Another way to synchronize the two threads is to use a second mailbox that sends a completion message from the Consumer back to the Producer, as shown in Sample 7.46.

Sample 7.46 Producer–consumer synchronized with a mailbox

```
program automatic mbx_mbx2;
  mailbox mbx, rtn;
  class Producer;
    task run();
      int k;
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
        rtn.get(k);
        $display("Producer: after  get(%0d)", k);
      end
    endtask
  endclass

  class Consumer;
    task run();
      int i;
      repeat (3) begin
        $display("Consumer: before get");
        mbx.get(i);
        $display("Consumer: after  get(%0d)", i);
        rtn.put(-i);
      end
    endtask
  endclass

  initial begin
    p = new();
    c = new();

    // Run the producer and consumer in parallel
    fork
      p.run();
      c.run();
    join
 end
endprogram
```

The return message in the `rtn` mailbox is just a negative version of the original integer. You could use any value, but this one can be checked against the original for debugging purposes.

Sample 7.47 Output from producer–consumer with mailbox

```
Producer: before put(1)
Consumer: before get
Consumer: after  get(1)
Consumer: before get
Producer: after  get(-1)
Producer: before put(2)
Consumer: after  get(2)
Consumer: before get
Producer: after  get(-2)
Producer: before put(3)
Consumer: after  get(3)
Producer: after  get(-3)
```

You can see from Sample 7.47 that the Producer and Consumer are successfully running in lockstep.

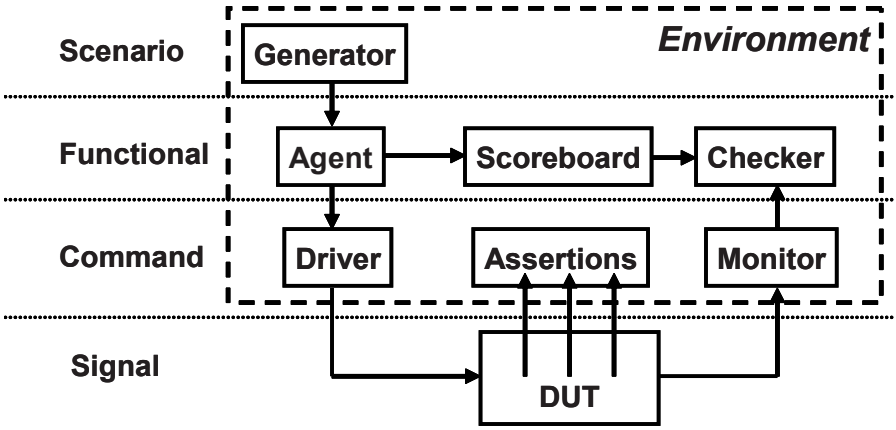
7.6.7 Other Synchronization Techniques

You can also complete the handshake by blocking on a variable or a semaphore. An event is the simplest construct, followed by blocking on a variable. A semaphore is comparable to using a second mailbox, but no information is exchanged. SystemVerilog's bounded mailbox just does not work as well as these other techniques, as there is no way to block the producer when it puts the first transaction in. Sample 7.42 shows that the Producer is always one transaction ahead of the Consumer.

7.7 Building a Testbench with Threads and IPC

Way back in Section 1.10 you learned about layered testbenches. Figure 7-8 shows the relationship between the different parts. Now that you know how to use threads and IPC, you can construct a basic testbench with transactors.

Figure 7-8 Layered testbench with environment



7.7.1 Basic Transactor

Sample 7.48 is the agent that sits between the Generator and the Driver.

Sample 7.48 Basic Transactor

```
class Agent;

  mailbox gen2agt, agt2drv;
  Transaction tr;

  function new(mailbox gen2agt, agt2drv);
    this.gen2agt = gen2agt;
    this.agt2drv = agt2drv;
  endfunction

  task run();
    forever begin
      gen2agt.get(tr);    // Get transaction from upstream block
      ...                 // Do some processing
      agt2drv.put(tr);    // Send it to downstream block
    end
  endtask

  task wrap_up();   // Empty for now
  endtask

endclass
```

7.7.2 Configuration Class

The configuration class allows you to randomize the configuration of your system for every simulation. Sample 7.49 has just one variable and a basic constraint.

Sample 7.49 Configuration class

```
class Config;
  bit [31:0] run_for_n_trans;
  constraint reasonable
    {run_for_n_trans inside {[1:1000]};
    }
endclass
```

7.7.3 Environment class

The Environment class, shown as a dashed line in Figure 7-8, holds the Generator, Agent, Driver, Monitor, Checker, Scoreboard, and Config objects, and the mailboxes between them. Sample 7.50 shows a basic Environment class.

Sample 7.50 Environment class

```
class Environment;

  Generator  gen;
  Agent      agt;
  Driver     drv;
  Monitor    mon;
  Checker    chk;
  Scoreboard scb;
  Config     cfg;
  mailbox gen2agt, agt2drv, mon2chk;

  extern function new();
  extern function void gen_cfg();
  extern function void build();
  extern task run();
  extern task wrap_up();
endclass

function Environment::new();
  cfg = new();
endfunction

function void Environment::gen_cfg;
  assert(cfg.randomize);
endfunction
```

```
function void Environment::build();
  // Initialize mailboxes
  gen2agt = new();
  agt2drv = new();
  mon2chk = new();

  // Initialize transactors
  gen = new(gen2agt);
  agt = new(gen2agt, agt2drv);
  drv = new(agt2drv);
  mon = new(mon2chk);
  chk = new(mon2chk);
  scb = new();
endfunction

task Environment::run();
  fork
    gen.run(cfg.run_for_n_trans);
    agt.run();
    drv.run();
    mon.run();
    chk.run();
    scb.run(cfg.run_for_n_trans);
  join
endtask

task Environment::wrap_up();
  fork
    gen.wrap_up();
    agt.wrap_up();
    drv.wrap_up();
    mon.wrap_up();
    chk.wrap_up();
    scb.wrap_up();
  join
endtask
```

Chapter 8 shows more details on how to build these classes.

7.7.4 Test Program

Sample 7.51 shows the main test, which is in a program block.

Sample 7.51 Basic test program

```
program automatic test;

  Environment env;

  initial begin
    env = new();
    env.gen_cfg();
    env.build();
    env.run();
    env.wrap_up();
  end
endprogram
```

7.8 Conclusion

Your design is modeled as many independent blocks running in parallel, and so your testbench must also generate multiple stimulus streams and check the responses using parallel threads. These are organized into a layered testbench, orchestrated by the top-level environment. SystemVerilog introduces powerful constructs such as `fork...join_none` and `fork...join_any` for dynamically creating new threads, in addition to the standard `fork...join`. These threads communicate and synchronize using events, semaphores, mailboxes, and the classic `@` event control and `wait` statements. Lastly, the `disable` command is used to terminate threads.

These threads and the related control constructs complement the dynamic nature of OOP. As objects are created and destroyed, they can run in independent threads, allowing you to build a powerful and flexible testbench environment.

Chapter 8

Advanced OOP and Testbench Guidelines

How would you create a complex class for a bus transaction that also performs error injection and has variable delays? The first approach is to put everything in a large, flat class. This approach is simple to build and easy to understand (all the code is right there in one class), but can be slow to develop and debug. Additionally, such a large class is a maintenance burden, as anyone who wants to make a new transaction behavior has to edit the same file. Just as you would never create a complex RTL design using just one Verilog module, you should break classes down into smaller, reusable blocks.

The next choice is composition. As you learned in Chap. 5, you can instantiate one class inside another, just as you instantiate modules inside another, building up a hierarchical testbench. You write and debug your classes from the top down or bottom up, always looking for natural partitions when deciding what variables and methods go into the various classes. A pixel could be partitioned into its color and coordinate. A packet might be divided into header and payload. You might break an instruction into opcode and operands. See Section 8.4 for guidelines on partitioning.

Sometimes it is difficult to divide the functionality into separate parts. Take the example of a bus transaction with error injection. When you write the original class for the transaction, you may not think of all the possible error cases. Ideally, you would like to make a class for a good transaction, and later add different error injectors. The transaction may have data fields and an error-checking CRC field generated from the data. One form of error injection is corruption of the CRC field. If you use composition,

you need separate classes for a good transaction, and an error transaction. Testbench code that used good objects would have to be rewritten to process the new error objects. What you need is a class that resembles the original class but adds a few new variables and methods. This result is accomplished through inheritance.

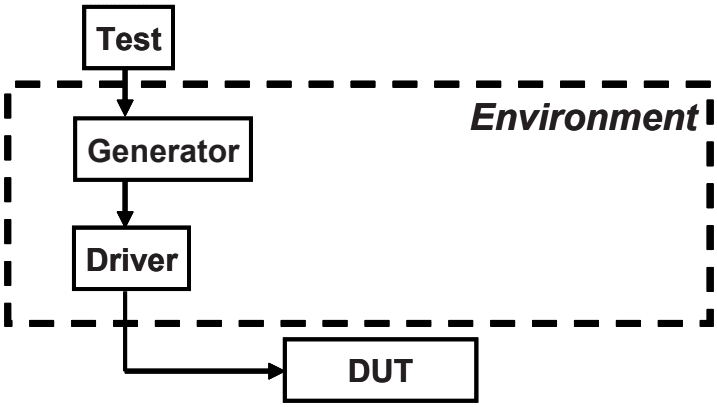
Inheritance allows a new class to be derived from an existing one in order to share its variables and routines. The original class is known as the base or super class. The new one, since it extends the capability of the base class, is called the extended class. Inheritance provides reusability by adding features, such as error injection, to an existing class, the basic transaction, without modifying the base class.

The real power of OOP is that it gives you the ability to take an existing class, such as a transaction, and selectively change parts of its behavior by replacing routines, but without having to change the surrounding infrastructure. With some planning, you can create a testbench solid enough to send basic transactions, but able to accommodate any extensions needed by the test.

8.1 Introduction to Inheritance

Figure 8-1 shows a simple testbench. A generator creates a transaction, randomizes it, and sends it to the driver. The rest of the testbench is left out.

Figure 8-1 Simplified layered testbench



8.1.1 Base Transaction

The base transaction class has variables for the source and destination addresses, eight data words, and a CRC for error checking, plus routines for displaying the contents and calculating the CRC. The `calc_crc` function is tagged as `virtual` so that it can be redefined if needed, as shown in the next section. Virtual routines are explained in more detail later in this chapter.

Sample 8.1 Base `Transaction` class

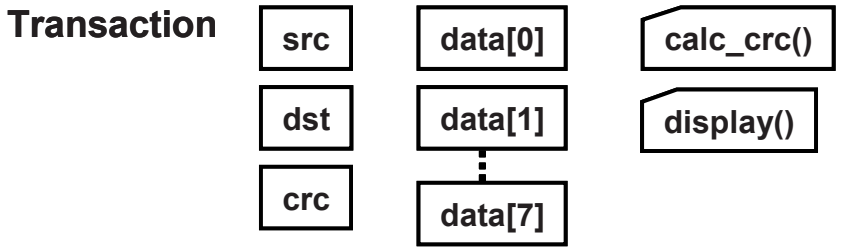
```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Random variables
  bit [31:0] crc;                     // Calculated variable

  virtual function void calc_crc;
    crc = src ^ dst ^ data.xor;
  endfunction

  virtual function void display(input string prefix="");
    $display("%sTr: src=%h, dst=%h, crc=%h",
             prefix, src, dst, crc);
  endfunction
endclass
```

Figure 8-2 shows both the variables and routines for the class.

Figure 8-2 Base `Transaction` class diagram



8.1.2 Extending the `Transaction` Class

Suppose you have a testbench that sends good transactions through the DUT and now want to inject errors. Take an existing transaction class and extend it to create a new class. If you follow the guidelines from Chap. 1, you would want to make as few code changes as possible to your existing testbench. So how can you reuse the existing `Transaction` class? This is done by declaring the new class, `BadTr`, as an extension of the current class. `Transaction` is called the base class, whereas `BadTr` is known as the extended class (Figure 8-3).

Sample 8.2 Extended `Transaction` class

```
class BadTr extends Transaction;
  rand bit bad_crc;

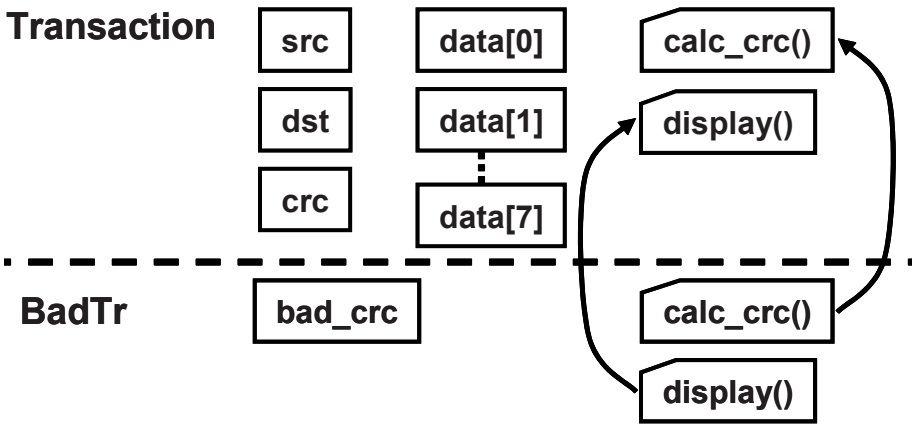
  virtual function void calc_crc;
    super.calc_crc();          // Compute good CRC
    if (bad_crc) crc = ~crc;  // Corrupt the CRC bits
  endfunction

  virtual function void display(input string prefix="");
    $write("%sBadTr: bad_crc=%b, ", prefix, bad_crc);
    super.display();
  endfunction

endclass : BadTr
```

Note that in Sample 8.2, the variable `crc` is used without a hierarchical identifier. The `BadTr` class can see all the variables from the original `Transaction` plus its own variables such as `bad_crc`. The `calc_crc` function in the extended class calls `calc_crc` in the base class using the `super` prefix. You can call one level up, but going across multiple levels such as `super.super.new` is not allowed in SystemVerilog. Not to mention that this style, since it reaches across multiple levels, violates the rules of encapsulation by reaching across multiple boundaries.

Figure 8-3 Extended `Transaction` class diagram





Always declare routines inside a class as virtual so that they can be redefined in an extended class. This applies to all tasks and functions, except the `new` function, which is called when the object is constructed, and so there is no way to extend it. SystemVerilog always calls the `new` function based on the handle's type.

8.1.3 More OOP Terminology

Here is a quick glossary of terms. As explained in Chap. 5, the OOP term for a variable in a class is "property," and a task or function is called a "method." When you extend a class, the original class (such as `Transaction`) is called the parent class or super class. The extended class (`BadTr`) is known as the derived class or sub class. A base class is one that is not derived from any other class. The "prototype" for a routine is just the first line that shows the argument list and return type, if any. The prototype is used when you move the body of the routine outside the class, but is needed to describe how the routine communicated with others, as shown in Section 5.11.

8.1.4 Constructors in Extended Classes

When you start extending classes, there is one rule about constructors (`new` function) to keep in mind. If your base class constructor has any arguments, the constructor in the extended class must have a constructor and must call the base's constructor on its first line.

Sample 8.3 Constructor with argument in an extended class

```
class Base1;
  int var;
  function new(input int var); // Has argument
    this.var = var;
  endfunction
endclass

class Extended extends Base1;
  function new(input int var); // Needs argument
    super.new(var);  // Must be first line of new
    // Other constructor actions
  endfunction
endclass
```

8.1.5 Driver Class

The following driver class receives transactions from the generator and drives them into the DUT.

Sample 8.4 Driver class

```
class Driver;
  mailbox gen2drv;

  function new(input mailbox gen2drv);
    this.gen2drv = gen2drv;
  endfunction

  task main;
    Transaction tr;         // Handle to a Transaction object or
                            // an class derived from Transaction
    forever begin
      gen2drv.get(tr);      // Get transaction from generator
      tr.calc_crc();        // Process the transation
      @ifc.cb.src = tr.src; // Send transaction
      ...
    end
  endtask
endclass
```

This class stimulates the DUT with `Transaction` objects. OOP rules say that if you have a handle of the base type (`Transaction`), it can also point to an object of an extended type (`BadTr`). This is because the handle `tr` can only reference the variables `src`, `dst`, `crc`, and `data`, and the routine `calc_crc`. So you can send `BadTr` objects into the driver without changing it.

See Chaps. 10 and 11 for examples of fully functional drivers with advanced features such as virtual interfaces and callbacks.

When the driver calls `tr.calc_crc`, which one will be called, the one in `Transaction` or `BadTr`? Since `calc_crc` was declared as a virtual method in Samples 8.1 and 8.2, SystemVerilog chooses the proper method based on the type of object stored in `tr`. If the object is of type `Transaction`, SystemVerilog calls the task `Transaction::calc_crc`. If it is of type `BadTr`, SystemVerilog calls `BadTr::calc_crc`.

8.1.6 Simple Generator Class

The generator for this testbench creates a random transaction and puts it in the mailbox to the driver. The following example shows how you might create the class from what you have learned so far. Note that this avoids a very common testbench bug by constructing a new transaction object every pass through the loop instead of just once outside. This bug is discussed in more detail in Section 7.6 on mailboxes.

Sample 8.5 Generator class

```
// Generator class that uses Transaction objects
// First attempt... too limited
class Generator;
  mailbox gen2drv;
  Transaction tr;

  function new(input mailbox gen2drv);
    this.gen2drv = gen2drv;    // this-> class-level var
  endfunction

  task run();
    forever begin
      tr = new();              // Construct transaction
      assert(tr.randomize());  // Randomize it
      gen2drv.put(tr);         // Send to driver
    end
  endtask
endclass
```

There is a problem with this generator. The `run` task constructs a transaction and immediately randomizes it. This means that the transaction uses whatever constraints are turned on by default. The only way you can change this would be to edit the `Transaction` class, which goes against the verification guidelines presented in this book. Worse yet, the generator only uses `Transaction` objects – there is no way to use an extended object such as `BadTr`. The fix is to separate the construction of `tr` from its randomization as shown below in Section 8.2.

As you build data-oriented classes such as network and bus transactions, you will see that they have common properties (`id`) and methods (`display`). Control-oriented classes such as the `Generator` and `Driver` classes also have a common structure. You can enforce this by having both of them the extensions of a base `Transactor` class, with virtual methods for `run` and `wrap_up`. The VMM has an extensive set of base classes for transactors, data, and much more.

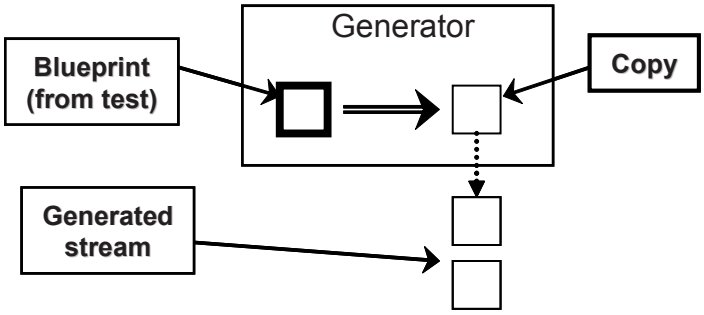
8.2 Blueprint Pattern



A useful OOP technique is the "blueprint pattern." If you have a machine to make signs, you don't need to know the shape of every possible sign in advance. You just need a stamping machine and then change the die to cut different shapes. Likewise, when you want to build a transactor generator, you don't have to know how to build every type of transaction; you just need to be able to stamp new ones that are similar to a given transaction.

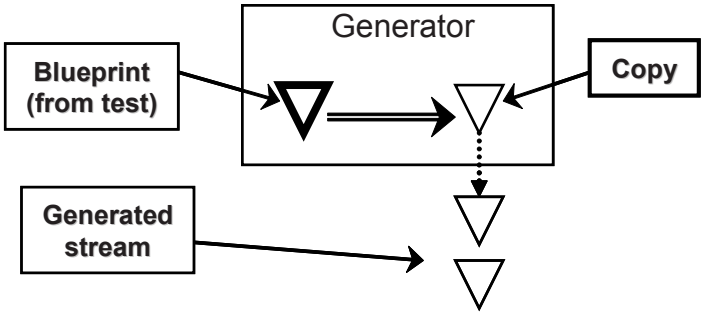
Instead of constructing and then immediately using an object, as in Sample 8.5, construct a blueprint object (the cutting die), and then modify its constraints, or even replace it with an extended object. Now when you randomize this blueprint, it will have the random values that you want. Make a copy of this object and send the copy to the downstream transactor.

Figure 8-4 Blueprint pattern generator



The beauty of this technique is that if you change the blueprint object, your generator creates a different-type object. Using the sign analogy, you change the cutting die from a square to a triangle to make Yield signs.

Figure 8-5 Blueprint generator with new pattern



The blueprint is the "hook" that allows you to change the behavior of the generator class without having to change its code. You need to make a copy method that can make a copy of the blueprint to transmit, so that the original blueprint object is kept around for the next pass through the loop (Figure 8-5).

Sample 8.6 shows the generator class using the blueprint pattern. The important thing to notice is that the blueprint object is constructed in one place (the `new` function) and used in another (the `run` task). Previous coding guidelines said to separate the declaration and construction; similarly, you need to separate the construction and randomization of the blueprint object.

Sample 8.6 Generator class using blueprint pattern

```
class Generator;
  mailbox gen2drv;
  Transaction blueprint;

  function new(input mailbox gen2drv);
    this.gen2drv = gen2drv;
    blueprint = new();
  endfunction

  task run();
    Transaction tr;
    forever begin
      assert(blueprint.randomize);
      tr = blueprint.copy();   // * see below
      gen2drv.put(tr);         // Send to driver
    end
  endtask
endclass
```

The `copy` method is discussed in Section 8.5. For now, remember that you must add it to the `Transaction` and `BadTr` classes. Sample 8.37 shows an advanced generator using templates.

8.2.1 The `Environment` Class

Chapter 1 discussed the three phases of execution: Build, Run, and Wrap-up. Sample 8.7 shows the environment class that instantiates all the testbench components, and runs these three phases.

Sample 8.7 `Environment` class

```
// Testbench environment class
class Environment;
  Generator gen;
  Driver drv;
  mailbox gen2drv;

  function void build();   // Build the environment by
    gen2drv = new();       // constructing the mailbox,
    gen = new(gen2drv);    // generator,
    drv = new(gen2drv);    // and driver
  endfunction

  task run();
    fork
      gen.run();
      drv.run();
    join_none
  endtask

  task wrap_up();
    // Empty for now - call scoreboard for report
  endtask

endclass
```

8.2.2 A Simple Testbench

The test is contained in the top-level program. The basic test just lets the environment run with all the defaults.

Sample 8.8 Simple test program using environment defaults

```
program automatic test;

  Environment env;
  initial begin
    env = new();          // Construct the environment
    env.build();          // Build testbench objects
    env.run();            // Run the test
    env.wrap_up;          // Clean up afterwards
  end
endprogram
```

8.2.3 Using the Extended `Transaction` Class



To inject an error, you need to change the blueprint object from a `Transaction` object to a `BadTr`. You do this between the build and run phases in the environment. The top-level testbench runs each phase of the environment and changes the blueprint. Note how all the references to `BadTr` are in this one file, and so you don't have to change the `Environment` or `Generator` classes. You want to restrict the scope of where `BadTr` is used, and so a standalone `begin...end` block is used in the middle of the `initial` block. This makes a visually distinctive block of code. You can take a shortcut and construct the extended class in the declaration.

Sample 8.9 Injecting an extended transaction into testbench

```
program automatic test;

  Environment env;
  initial begin
    env = new();
    env.build();                 // Construct generator, etc.

    begin
      BadTr bad = new();         // Replace blueprint with
      env.gen.blueprint = bad;   // the "bad" one
    end

    env.run();                   // Run the test with BadTr
    env.wrap_up();               // Clean up afterwards
  end
endprogram
```

8.2.4 Changing Random Constraints with an Extended Class



In Chap. 6, you learned how to generate constrained random data. Most of your tests are going to need to further constrain the data, which is best done with inheritance. In Sample 8.10, the original Transaction class is extended to include a new constraint that keeps the destination address in the range of ±100 of the source address.

Sample 8.10 Using inheritance to add a constraint

```
class Nearby extends Transaction;
  constraint c_nearby {
    dst inside {[src-100:src+100]};
  }
endclass


program automatic test;
  Environment env;
  initial begin
    env = new();
    env.build();                // Construct generator, etc.

    begin
      Nearby nb = new();        // Create a new blueprint
      env.gen.blueprint = nb;   // Replace the blueprint
    end

    env.run();                  // Run the test with Nearby
    env.wrap_up();              // Clean up afterwards
  end
endprogram
```

Note that if you define a constraint in an extended class with the same name as one in the base class, the extended constraint replaces the base one. This allows you to change the behavior of existing constraints.

8.3 Downcasting and Virtual Methods

As you start to use inheritance to extend the functionality of classes, you need a few OOP techniques to control the objects and their functionality. In particular, a handle can refer to an object for a certain class, or any extended class. So what happens when a base handle points to an extended object? What happens when you call a method that exists in both the base and extended classes? This section explains what happens using several examples.

8.3.1 Downcasting with `$cast`

Downcasting or conversion is the act of casting a pointer to a base class to one to a derived class. Consider the base class and extended class in Sample 8.11 and Figure 8-6.

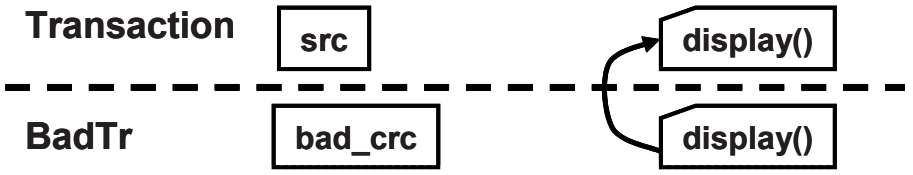
Sample 8.11 Base and extended class

```
class Transaction;
  rand bit [31:0] src;
  virtual function void display(input string prefix="");
    $display("%sTransaction: src=%0d", prefix, src);
  endfunction
endclass

class BadTr extends Transaction;
  bit bad_crc;
  virtual function void display(input string prefix="");
    $display("%sBadTr: bad_crc=%b", prefix, bad_crc);
    super.display(prefix);
  endfunction
endclass

Transaction tr;
BadTr bad, bad2;
```

Figure 8-6 Simplified extended transaction



You can assign an extended handle to a base handle, and no special code is needed, as shown in Sample 8.12.

Sample 8.12 Copying extended handle to base handle

```
Transaction tr;
BadTr bad;
bad = new();        // Construct BadTr extended object
tr = bad;           // Base handle points to extended obj
$display(tr.src);   // Display base variable
tr.display;         // Calls BadTr::display
```

When a class is extended, all the base class variables and methods are included, and so the integer `src` exists in the extended object. The assignment on the second line is permitted, as any reference using the base handle `tr` is valid, such as `tr.src` and `tr.display`.

What if you try going in the opposite direction, copying a base object into an extended handle, as shown in Sample 8.13? This fails because the base object is missing properties that only exist in the extended class, such as `bad_crc`. The

SystemVerilog compiler does a static check of the handle types and will not compile the second line.

Sample 8.13 Copying a base handle to an extended handle

```
tr = new();              // Construct base object
bad = tr;                // ERROR: WILL NOT COMPILE
$display(bad.bad_crc);   // bad_crc is not in base object
```

It is not always illegal to assign a base handle to an extended handle. It is allowed when the base handle actually points to an extended object. The `$cast` routine checks the type of object referenced by the handles, not just the handle. If the source object is the same type as the destination, or a class extended from the destination's class, you can copy the address of the extended object from the base handle, `tr`, into the extended handle, `bad2`.

Sample 8.14 Using `$cast` to copy handles

```
bad = new();          // Construct BadTr extended object
tr = bad;             // Base handle points to extended obj

// Check the object type & copy. Simulation error if mismatch
// If successful, bad2 points to the object referenced by tr
$cast(bad2, tr);

// Check for type mismatch, no simulation error
if(!$cast(bad2, tr))
  $display("cannot assign tr to bad2");

$display(bad2.bad_crc); // bad_crc exists in original obj
```

When you use `$cast` as a task, SystemVerilog checks the type of the source object at run-time and gives an error if it is not compatible with the destination. When you use `$cast` as a function, SystemVerilog still checks the type, but no longer prints an error if there is a mismatch. The `$cast` function returns zero when the types are incompatible, and non-zero for compatible types.

8.3.2 Virtual Methods

By now you should be comfortable using handles with extended classes. What happens if you try to call a routine using one of these handles?

Sample 8.15 `Transaction` and `BadTr` classes

```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Variables
  bit [31:0] crc;

  virtual function void calc_crc();   // XOR all fields
    crc = src ^ dst ^ data.xor;
  endfunction
endclass : Transaction

class BadTr extends Transaction;
  rand bit bad_crc;
  virtual function void calc_crc();
    super.calc_crc();           // Compute good CRC
    if (bad_crc) crc = ~crc;    // Corrupt the CRC bits
  endfunction
endclass : BadTr
```

Here is a block of code that uses handles of different types.

Sample 8.16 Calling class methods

```
Transaction tr;
BadTr bad;

initial begin
  tr = new();
  tr.calc_crc();   // Calls Transaction::calc_crc

  bad = new();
  bad.calc_crc();  // Calls BadTr::calc_crc

  tr = bad;        // Base handle points to ext obj
  tr.calc_crc();   // Calls BadTr::calc_crc
end
```

To decide which virtual method to call, SystemVerilog uses the object's type, not the handle's type. In the last statement of Sample 8.16, `tr` points to an extended object (`BadTr`) and so `BadTr::calc_crc` is called.

If you left out the `virtual` modifier on `calc_crc`, SystemVerilog would use the type of the handle `tr` (`Transaction`), not the object. That last statement would call `Transaction::calc_crc` – probably not what you wanted.

The OOP term for multiple routines sharing a common name is "polymorphism." It solves a problem similar to what computer architects faced when trying to make a processor that could address a large address space but had only a small amount of physical memory. They created the concept of virtual memory, where the code and data for a program could reside in memory or on a disk. At compile time, the program

didn't know where its parts resided – that was all taken care of by the hardware plus operating system at run-time. A virtual address could be mapped to some RAM chips, or the swap file on the disk. Programmers no longer needed to worry about this virtual memory mapping when they wrote code – they just knew that the processor would find the code and data at run-time. See also Denning (2005).

8.3.3 Signatures

There is one downside to using virtual methods – once you define one, all extended classes that define the same virtual routine must use the same "signature," i.e., the same number and type of arguments. You cannot add or remove an argument in an extended virtual routine. This just means you need to plan ahead.

There is a good reason that SystemVerilog and other OOP languages require that a virtual method must have the same signature as the one in the parent (or grandparent). If you were able to add an additional argument, or turn a task into a function, polymorphism would no longer work. Your code needs to be able to call a virtual method with the assurance that a method in a derived class will have the same interface.

8.4 Composition, Inheritance, and Alternatives

As you build up your testbench, you have to decide how to group related variables and routines together into classes. In Chap. 5, you learned how to build basic classes and include one class inside another. Previously in this chapter, you saw the basics of inheritance. This section shows you how to decide between the two styles, and also shows an alternative.

8.4.1 Deciding Between Composition and Inheritance

How should you tie together two related classes? Composition uses a "has-a" relationship. A packet has a header and a body. Inheritance uses an "is-a" relationship. A `BadTr` is a `Transaction`, just with more information. Table 8-1 is a quick guide, with more details.

Table 8-1 Comparing inheritance to composition

Question	Inheritance	Composition
Do you need to group multiple subclasses together? (SystemVerilog does not support multiple inheritance)	No	Yes
Does the higher-level class represent objects at a similar level of abstraction?	Yes	No
Is the lower-level information always present or required?	Yes	No
Does the additional data need to remain attached to the original class while it is being processed by pre-existing code?	Yes	No

1. Are there several small classes that you want to combine into a larger class? For example, you may have a data class and header class and now want to make a packet class. SystemVerilog does not support multiple inheritance, where one class derives from several classes at once. Instead you have to use composition. Alternatively, you could extend one of the classes to be the new class, and manually add the information from the others.
2. In Sample 8.15, the `Transaction` and `BadTr` classes are both bus transactions that are created in a generator and driven into the DUT. Thus inheritance makes sense.
3. The lower-level information such as `src`, `dst`, and `data` must always be present for the Driver to send a transaction.
4. In Sample 8.15, the new `BadTr` class has a new field `bad_crc` and the extended `calc_crc` function. The `Generator` class just transmits a transaction and does not care about the additional information. If you use composition to create the error bus transaction, the `Generator` class would have to be rewritten to handle the new type.

If two objects seem to be related by both "is-a" and "has-a," you may need to break them down into smaller components.

8.4.2 Problems with Composition

The classical OOP approach to building a class hierarchy partitions functionality into small blocks that are easy to understand. However, testbenches are not standard software development projects, as was discussed in Section 5.16 on public vs. local attributes. Concepts such as information hiding (using local variables) conflict with

building a testbench that needs maximum visibility and controllability. Similarly, dividing a transaction into smaller pieces may cause more problems than it solves.

When you are creating a class to represent a transaction, you may want to partition it to keep the code more manageable. For example, you may have an Ethernet MAC frame and your testbench uses two flavors, normal (`II`) and Virtual LAN (`VLAN`). Using composition, you could create a basic cell `EthMacFrame` with all the common fields such as `da` and `sa` and a discriminant variable, `kind`, to indicate the type. There is a second class to hold the VLAN information, which is included in `EthMacFrame`.

Sample 8.17 Building an Ethernet frame with composition

```
// Not recommended
class EthMacFrame;
  typedef enum {II, IEEE} kind_e;
  rand kind_e kind;
  rand bit [47:0] da, sa;
  rand bit [15:0] len;

  ...
  rand Vlan vlan_h;
endclass

class Vlan;
  rand bit [15:0] vlan;
endclass
```

There are several problems with composition. First, it adds an extra layer of hierarchy, and so you have to constantly add an extra name to every reference. The VLAN information is called `eth_h.vlan_h.vlan`. If you start adding more layers, the hierarchical names become a burden.

A more subtle issue occurs when you want to instantiate and randomize the hierarchy of classes. What does the `EthMacFrame` constructor create? Since `kind` is random, you don't know whether to construct a `Vlan` object when `new` is called. When you randomize the class, the constraints set variables in both the `EthMacFrame` and `Vlan` objects based on the random `kind` field. You have a circular dependency in that randomization only works on objects that have been instantiated, but you can't instantiate these objects until `kind` has been chosen.

The only solution to the construction and randomization problems is to always instantiate all objects in `EthMacFrame::new`. However, if you are always using all alternatives, why divide the Ethernet cell into two different classes?

8.4.3 Problems with Inheritance

Inheritance can solve some of these issues. Variables in the extended classes can be referenced without the extra hierarchy as in `eth_h.vlan`. You don't need a discriminant, but you may find it easier to have one variable to test rather than doing type-checking.

Sample 8.18 Building an Ethernet frame with inheritance

```
// Not recommended
class EthMacFrame;
  typedef enum {II, IEEE} kind_e;
  rand kind_e kind;
  rand bit [47:0] da, sa;
  rand bit [15:0] len;
  ...
endclass

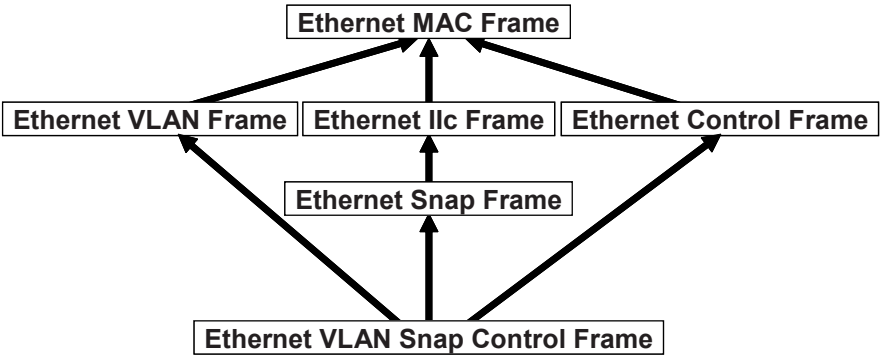
class Vlan extends EthMacFrame;
  rand bit [15:0] vlan;
endclass
```

On the downside, a set of classes that use inheritance always requires more effort to design, build, and debug than a set of classes without inheritance. Your code must use `$cast` whenever you have an assignment from a base handle to an extended. Building a set of virtual methods can be challenging, as they all have to have the same prototype. If you need an extra argument, you need to go back and edit the entire set, and possibly the routine calls too.

There are also problems with randomization. How do you make a constraint that randomly chooses between the two kinds of frame and sets the proper variables? You can't put a constraint in `EthMacFrame` that references the `vlan` field.

The final issue is with multiple inheritance. In Figure 8-7, you can see how the VLAN frame is derived from a normal MAC frame. The problem is that these different standards reconverged. SystemVerilog does not support multiple inheritance, and so you could not create the VLAN/Snap/Control frame through inheritance.

Figure 8-7 Multiple inheritance problem



8.4.4 A Real-World Alternative

If composition leads to large hierarchies, but inheritance requires extra code and planning to deal with all the different classes, and both have difficult construction and randomization, what can you do? You can instead make a single, flat class that has all the variables and routines. This approach leads to a very large class, but it handles all the variants cleanly. You have to use the discriminant variable often to tell which variables are valid, as shown in Sample 8.19. It contains several conditional constraints, which apply in different cases, depending on the value of `kind`.

Sample 8.19 Building a flat Ethernet frame

```
class eth_mac_frame;
  typedef enum {II, IEEE} kind_e;
  rand kind_e kind;
  rand bit [47:0] da, sa;
  rand bit [15:0] len, vlan;
  ...
  constraint eth_mac_frame_II {
    if (kind == II) {
      data.size() inside {[46:1500]};
      len == data.size();
  }}
  constraint eth_mac_frame_ieee {
    if (kind == IEEE) {
      data.size() inside {[46:1500]};
      len < 1522;
  }}
endclass
```

Regardless of how you build your classes, you should define the typical behavior and constraints in the class, and then use inheritance to inject new behavior at the test level.

8.5 Copying an Object

In Sample 8.6, the generator first randomized, and then copied the blueprint to make a new transaction. Take a closer look at the `copy` function in Sample 8.20.

Sample 8.20 Base transaction class with a virtual `copy` function

```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Variables
  bit [31:0] crc;

  virtual function Transaction copy();
    copy = new();
    copy.src = src;  // Copy data fields
    copy.dst = dst;
    copy.data = data;
    copy.crc = crc;
  endfunction
endclass
```

When you extend the `Transaction` class to make the class `BadTr`, the `copy` function still has to return a `Transaction` object. This is because the extended virtual function must match the base `Transaction::copy`, including all arguments and return type, as shown in Sample 8.21

Sample 8.21 Extended transaction class with virtual `copy` method

```
class BadTr extends Transaction;
  rand bit bad_crc;

  virtual function Transaction copy();
    BadTr bad;
    bad = new();
    bad.src = src;  // Copy data fields
    bad.dst = dst;
    bad.data = data;
    bad.crc = crc;
    bad.bad_crc = bad_crc;
    return bad;
  endfunction

endclass : BadTr
```

8.5.1 The `copy_data` method

One optimization is to break the `copy` function in two, creating a separate function, `copy_data`. Now each class is responsible for copying its local data. This makes the `copy` function more robust and reusable. Here is the function for the base class.

Sample 8.22 Base transaction class with `copy_data` function

```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Variables
  bit [31:0] crc;

  virtual function void copy_data(input Transaction tr);
    tr.src  = src;    // Copy the data fields
    tr.dst  = dst;
    tr.data = data;
    tr.crc  = crc;
  endfunction

  virtual function Transaction copy();
    copy = new();
    copy_data(copy);
  endfunction
endclass
```

In the extended class, the `copy_data` method is a little more complicated. Since it extends the original `copy_data`, it must have a single argument, a `Transaction` handle. The method can use this handle when calling `Transaction::copy_data`, but when `copy_data` needs to copy `bad_crc`, it needs a `BadTr` handle, and so it has to first cast the base handle to the extended type, as shown in Sample 8.23.

Sample 8.23 Extended transaction class with `copy_data` function

```
class BadTr extends Transaction;
  rand bit bad_crc;

  virtual function void copy_data(input Transaction tr);
    BadTr bad;
    super.copy_data(tr);      // Copy base data
    $cast(bad, tr);           // Cast base handle to extÕd
    bad.bad_crc = bad_crc;    // Copy extended data
  endfunction

  virtual function Transaction copy();
    BadTr bad;
    bad = new();              // Construct BadTr
    copy_data(bad);           // Copy data fields
    return bad;
  endfunction
endclass : BadTr
```

8.5.2 Specifying a Destination for Copy

The existing `copy` routine always constructs a new object. An improvement for `copy` is to specify the location where the copy should be put. This technique is useful when you want to reuse an existing object, and not allocate a new one.

Sample 8.24 Base transaction class with `copy` function

```
class Transaction;

  virtual function Transaction copy(Transaction to=null);
    if (to == null)
      copy = new();// Construct new object
    else
      copy = to;   // or use exisiting
    copy_data(copy);
  endfunction

  // Uses copy_data() method from Sample 8.22
endclass
```

The only difference is the additional argument to specify the destination, and the code to test that a destination object was passed to this routine. If nothing was passed (the default), construct a new object, or else use the existing one.

Since you have added a new argument to a virtual method in the base class, you will have to add it to the same method in the extended classes, such as `BadTr`.

Sample 8.25 Extended transaction class with new `copy` function

```
class BadTr;

  virtual function Transaction copy(Transaction to=null);
    BadTr bad;
    if (to == null)
      bad = new();              // Create a new object
    else
      assert($cast(bad, to));   // Reuse existing one
    copy_data(bad);             // Copy data fields
    return bad;
  endfunction

endclass : BadTr
```

8.6 Abstract Classes and Pure Virtual Methods

By now you have seen classes with methods to perform common operations such as copying and displaying. One goal of verification is to create code that can be shared across multiple projects. If you can get your company to agree on common set of methods, it is easier to reuse code.

OOP languages such as SystemVerilog have two constructs to allow you to build a shareable base class. The first are an abstract class, which is a class that can be extended, but not instantiated directly. It is defined with the `virtual` keyword. Second are pure virtual methods, which are a prototype without a body. A class extended from an abstract class can only be instantiated if all virtual methods have bodies. The pure keyword specifies that a method declaration is a prototype, and not just an empty virtual method. Lastly, you can only declare pure virtual methods in an abstract class. You can declare non-pure methods in an abstract class. Note that the LRM allows you to define a virtual method without a body - you can call it but it just immediately returns.

Sample 8.26 shows an abstract class, `BaseTr`, which is a base class for transactions. It starts with some useful properties such as `id` and `count`. The constructor makes sure every instance has a unique ID. Next are pure virtual methods to compare, copy, and display the object.

Sample 8.26 Abstract class with pure virtual methods

```
virtual class BaseTr;
  static int count;     // Number of instance created
  int id;               // Unique transaction id

  function new();
    id = count++;       // Give each object a unique ID
  endfunction

  pure virtual function bit compare(input BaseTr to);
  pure virtual function BaseTr copy(input BaseTr to=null);
  pure virtual function void display(input string prefix="");

endclass : BaseTr
```

You can declare handles of type `BaseTr`, but you cannot construct objects of this type. You need to extend the class and provide implementations for all the pure virtual methods.

Sample 8.27 shows the definition of the `Transaction` class, which has been extended from `BaseTr`. Since `Transaction` has bodies for all the pure virtual methods, you can use it in your testbench.

Sample 8.27 Transaction class extends abstract class

```
class Transaction extends BaseTr;
  rand bit [31:0] src, dst, crc, data[8];

  extern virtual function bit compare(input BaseTr to);
  extern virtual function BaseTr copy(input BaseTr to=null);
  extern virtual function void copy_data
                                (input Transaction copy);
  extern virtual function void display (input string prefix="");
  extern function new();
endclass
```

Sample 8.28 Bodies for Transaction methods

```
function bit Transaction::compare(input BaseTr to);
  Transaction tr;
  assert($cast(tr, to));  // Check if ÕtoÕ is correct type
  return ((this.src == tr.src) &&
          (this.dst == tr.dst) &&
          (this.crc == tr.crc) &&
          (this.data == tr.data));
endfunction : compare

function BaseTr Transaction::copy(input BaseTr to=null);
  Transaction cp;
  if (to == null) cp = new();
  else            $cast(cp, to);
  copy_data(cp);
  return cp;
  endfunction

function void Transaction::copy_data(Transaction copy);
  copy.src  = src;    // Copy the data fields
  copy.dst  = dst;
  copy.data = data;
  copy.crc  = crc;
endfunction

function void Transaction::display(string prefix="");
  $display("%sTransaction %0d src=%h, dst=%x, crc=%x",
           prefix, id, src, dst, crc);
endfunction : display;

function Transaction::new();
  super.new();
endfunction : new
```

Abstract classes and pure virtual methods let you build testbenches that have a common look and feel. This allows any engineer to read your code and quickly understand the structure.

8.7 Callbacks

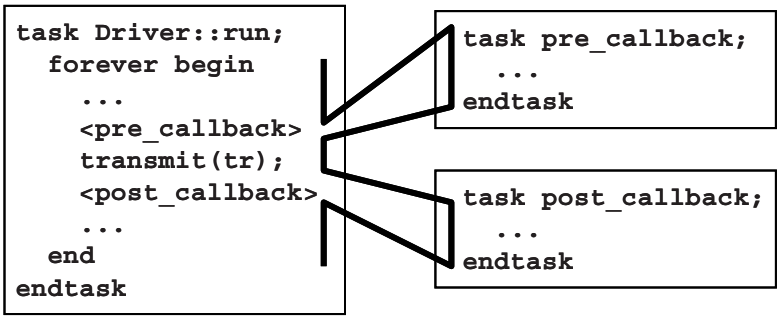
One of the main guidelines of this book is to create a single verification environment that you can use for all tests with no changes. The key requirement is that this test-

bench must provide a "hook," where the test program can inject new code without modifying the original classes. Your driver may want to do the following.

- Inject errors
- Drop the transaction
- Delay the transaction
- Synchronize this transaction with others
- Put the transaction in the scoreboard
- Gather functional coverage data

Rather than trying to anticipate every possible error, delay, or disturbance in the flow of transactions, the driver just needs to "call back" a routine that is defined in the top-level test. The beauty of this technique is that the callback[1] routine can be defined differently in every test. As a result, the test can add new functionality to the driver using callbacks without editing the `Driver` class.

Figure 8-8 Callback flow



In Figure 8-8, the `Driver::run` task loops forever with a call to a `transmit` task. Before sending the transaction, `run` calls the pretransmit callback, if any. After sending the transaction, it calls the postcallback task, if any. By default, there are no callbacks, and so `run` just calls `transmit`.

You could make `Driver::run` a virtual method and then override its behavior in an extended class, perhaps `MyDriver::run`. The drawback to this is that you might have to duplicate all the original method's code in the new method if you are injecting new behavior. Now if you made a change in the base class, you would have to remember to propagate it to all the extended classes. Additionally, you can inject a callback without modifying the code that constructed the original object.

[1]This callback technique is not related to Verilog PLI callbacks or SVA callbacks.

8.7.1 Creating a Callback

A callback task is created in the top-level test and called from the driver, the lowest level of the environment. However, the driver does not have to have any knowledge of the test – it just has to use a generic class that the test can extend. The driver uses a queue to hold the callback objects, which allows you to add multiple objects. The base callback class is an abstract class that must be extended before being used.

Sample 8.29 Base callback class

```
virtual class Driver_cbs;  // Driver callbacks
 virtual task pre_tx(ref Transaction tr, ref bit drop);
    // By default, callback does nothing
  endtask

  virtual task post_tx(ref Transaction tr);
    // By default, callback does nothing
  endtask
endclass
```

Sample 8.30 Driver class with callbacks

```
class Driver;
  Driver_cbs cbs[$];

  task run();
    bit drop;
    Transaction tr;

    forever begin
      drop = 0;
      agt2drv.get(tr);
      foreach (cbs[i]) cbs[i].pre_tx(tr, drop);
      if (!drop) continue;

      transmit(tr);

      foreach (cbs[i]) cbs[i].post_tx(tr);
    end
  endtask
endclass
```

Note that while `Driver_cbs` is an abstract class, `pre_tx` and `post_tx` are not pure virtual methods. This is because a typical callback uses only one of them. If a class has even one pure virtual method with an implementation, OOP rules won't allow you to instantiate it.

8.7.2 Using a Callback to Inject Disturbances

A common use for a callback is to inject some disturbance such as causing an error or delay. The following testbench randomly drops packets using a callback object. Callbacks can also be used to send data to the scoreboard or to gather functional coverage values. Note that you can use `push_back()` or `push_front()` depending on the order in which you want these to be called. For example, you probably want the scoreboard called after any tasks that may delay, corrupt, or drop a transaction. Only gather coverage after a transaction has been successfully transmitted.

Sample 8.31 Test using a callback for error injection

```
class Driver_cbs_drop extends Driver_cbs;

 virtual task pre_tx(ref Transaction tr, ref bit drop);
    // Randomly drop 1 out of every 100 transactions
    drop = ($urandom_range(0,99) == 0);
  endtask

endclass


program automatic test;

  Environment env;

  initial begin
    env = new();
    env.gen_cfg();
    env.build();

    begin              // Create error injection callback
      Driver_cbs_drop dcd = new();
      env.drv.cbs.push_back(dcd); // Put into driverÕs Q
    end

    env.run();
    env.wrap_up();
  end

endprogram
```

8.7.3 A Quick Introduction to Scoreboards

The design of your scoreboard depends on the design under test. A DUT that processes atomic transactions such as packets may have a scoreboard that contains a transform function to turn the input transactions into expected values, a memory to

hold these values, and a compare routine. A processor design needs a reference model to predict the expected output, and the comparison between expected and actual values may happen at the end of simulation.

Sample 8.32 shows a simple scoreboard that stores transactions in a queue of expected values. The first method saves an expected transaction, and the second tries to find an expected transaction that matches an actual one that was received by the testbench. Note that when you search through a queue, you can get 0 matches (transaction not found), 1 match (ideal case), or multiple matches (you need to do a more sophisticated match).

Sample 8.32 Simple scoreboard for atomic transactions

```
class Scoreboard;
  Transaction scb[$];  // Store expected trÕs in queue

  function void save_expect(input Transaction tr);
    scb.push_back(tr);
  endfunction

  function void compare_actual(input Transaction tr);
    int q[$];

    q = scb.find_index(x) with (x.src == tr.src);
    case (q.size())
      0: $display("No match found");
      1: scb.delete(q[0]);
      default:
        $display("Error, multiple matches found!");
    endcase
   endfunction : compare_actual
endclass : Scoreboard
```

8.7.4 Connecting to the Scoreboard with a Callback

The following testbench creates its own extension of the driver's callback class and adds a reference to the driver's callback queue. Note that the scoreboard callback needs a handle to the scoreboard and so it can call the method to save the expected transaction. This example does not show the monitor side, which will need its own callback to send the actual transaction to the scoreboard for comparison.

Sample 8.33 Test using callback for scoreboard

```
class Driver_cbs_scoreboard extends Driver_cbs;
  Scoreboard scb;

 virtual task pre_tx(ref Transaction tr, ref bit drop);
    // Put transaction in the scoreboard
    scb.save_expected(tr);
  endtask

  function new(input Scoreboard scb);
    this.scb = scb;
  endfunction
endclass


program automatic test;

  Environment env;

  initial begin
    env = new();
    env.gen_cfg();
    env.build();

    begin                    // Create scoreboard callback
      Driver_cbs_scoreboard dcs = new(env.scb);
      env.drv.cbs.push_back(dcs); // Put into driverÕs Q
    end

    env.run();
    env.wrap_up();
  end

endprogram
```

Always use callbacks for scoreboards and functional coverage. The monitor transactor can use a callback to compare received transactions with expected ones. The monitor callback is also the perfect place to gather functional coverage on transactions that are actually sent by the DUT.

You may have thought of putting the scoreboard or functional coverage group in a transactor, and connect it to the testbench using a mailbox. This is a poor solution for several reasons. These testbench components are almost always passive and asynchronous, and so they only wake up when the testbench has data for them, plus they never pass information to a downstream transactor. Thus a transactor that has to monitor multiple mailboxes concurrently is an overly complex solution. Additionally, you may sample data from several points in your testbench, but a transactor is designed

for a single source. Instead, put methods in your scoreboard and coverage classes to gather data, and connect them to the testbench with callbacks.

8.7.5 Using a Callback to Debug a Transactor

If a transactor with callbacks is not working as expected, you can use an additional callback to debug it. You can start by adding a callback to display the transaction. If there are multiple instances of the transactor, display the hierarchical path, using `$display("%m")`. Then put debug code before and after the other callbacks to locate the one that is causing the problem. Even for debug, you have to avoid making changes to the testbench environment.

8.8 Parameterized Classes

As you become more comfortable with classes, you may notice that a data structure that performs a set of actions, such as a stack or generator, only works on a single data type. This sections shows how you can define a single class that works with multiple data types.

8.8.1 A Simple Stack

A common data structure is a stack, which has `push` and `pop` methods to store and retrieve data. Sample 8.34 shows a simple stack that works with integers.

Sample 8.34 Stack using the int type

```
class IntStack;
  local int stack[100];            // Holds data values
  local int top;

  function void push(input int i); // Push value on top
    stack[++top] = i;
  endfunction : push

  function int pop();             // Remove value from top
    return stack[top--];
  endfunction

endclass : IntStack
```

The problem with this stack is that it only works with integers. If you want to make a stack for real numbers, you would have to copy the class, and change the data type from `int` to `real`. This quickly leads to a proliferation of classes, which can become a maintenance problem if you ever want to add new operations such as traversing or printing the stack contents.

In SystemVerilog you can add a data type parameter to a class and then specify a type when you declare handles to that class. This is similar to, but more powerful than, a parameterized module, where you can specify a value such as bus width when it is instantiated. SystemVerilog's parameterized classes are similar to templates in C++.

Sample 8.35 is a parameterized class for a stack. Notice how the type `T` is defined on the first line with a default type of `int`.

Sample 8.35 Parameterized class for a stack

```
class Stack #(type T=int);
  local T stack[100];            // Holds data values
  local int top;

  function void push(input T i); // Push new value on top
    stack[++top] = i;
  endfunction : push

  function T pop();          // Remove value from top
    return stack[top--];
  endfunction

endclass : Stack
```

Sample 8.36 creates a stack for real numbers, and writes and reads values.

Sample 8.36 Using the parameterized stack class

```
initial begin
  Stack #(real) rStack;     // Create a real stack

  rStack = new();
  for(int i=0; i<5; i++)
    rStack.push(i*2.0);     // Push values onto stack

  for(int i=0; i<5; i++)
    $display("%f ",
             rStack.pop()); // Pop values off stack
end
```

Atomic generators are a great example of a class that can be parameterized. Once you have defined the class for one, the same structure works for any data type. Sample 8.37 takes the atomic generator from Sample 8.6 and adds a parameter so that you can generate any random object. The generator should be part of a package of verification classes. It needs to specify a the default type, and so it uses `BaseTr` from Sample 8.26, as this abstract class should also be part of the verification package.

Sample 8.37 Parameterized generator class using blueprint pattern

```
class Generator #(type T=BaseTr);
  mailbox gen2drv;
  T blueprint;                      // Blueprint object

  function new(input mailbox gen2drv);
    this.gen2drv = gen2drv;
    blueprint = new();              // Create default
  endfunction

  task run();
    T tr;
    forever begin
      assert(blueprint.randomize); // Randomize object
      tr = blueprint.copy();       // Make a copy
      gen2drv.put(tr);             // Send to driver
    end
  endtask
endclass
```

Using the `Transaction` class from Samples 8.27 and 8.28 and the generator on this page, you can build a simple testbench. It starts the generator and prints the first five transactions, using the mailbox synchronization shown in Sample 7.41.

Sample 8.38 Simple testbench using parameterized generator class

```
program automatic test;

  initial begin
    Generator #(Transaction) gen;
    mailbox gen2drv;
    gen2drv = new(1);
    gen = new(gen2drv);

    fork
      gen.run();

      repeat (5) begin
        Transaction tr;
        gen2drv.peek(tr);  // Get next transaction
        tr.display();
        gen2drv.get(tr);   // Remove transaction
      end

     join_any

    end
endprogram // test
```

8.8.2 Parameterized Class Suggestions

When creating parameterized classes, you should start with a nonparameterized class, debug it thoroughly, and then add parameters. This separation reduces your debug effort.

Macros are an alternative to parameterized classes. For example, you could define a macro for the generator and pass it the transaction data type. Macros are harder to debug than parameterized classes.

If you need to define several related classes that all share the same transaction type, you could use parameterized classes or a single large macro. In the end, how you define your classes is not as important as what goes into them.

A common set of virtual methods in your transaction class help you when creating parameterized classes. The `Generator` class uses the `copy` method, knowing that it always has the same signature. Likewise, the `display` method allows you to easily debug transactions as they flow through your testbench components.

8.9 Conclusion

The software concept of inheritance, where new functionality is added to an existing class, parallels the hardware practice of extending the design's features for each generation, while still maintaining backwards compatibility.

For example, you can upgrade your PC by adding a larger capacity disk. As long as it uses the same interface as the old one, you do not have to replace any other part of the system, yet the overall functionality is improved.

Likewise, you can create a new test by "upgrading" the existing driver class to inject errors. If you use an existing callback in the driver, you do not have to change any of the testbench infrastructure.

You need to plan ahead if you want use these OOP techniques. By using virtual routines and providing sufficient callback points, your test can modify the behavior of the testbench without changing its code. The result is a robust testbench that does not need to anticipate every type of disturbance (error-injection, delays, synchronization) that you may want as long as you leave a hook where the test can inject its own behavior.

The testbench is more complex than what you have previously constructed, but there is a payback in that the tests become smaller and easier to write. The testbench does the hard work of sending stimulus and checking responses, and so the test only has to make small tweaks to cause specialized behavior. An extra few lines of testbench code might replace code that would have to be repeated in every single test.

Lastly, OOP techniques improve your productivity by allowing you to reuse classes. For example, a parameterized class for a stack that operates on any other class, rather than a single type, saves you from having to create duplicate code.

Chapter 9

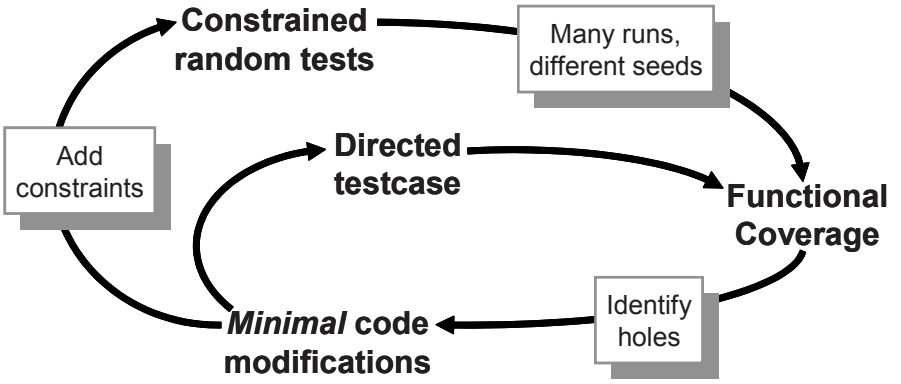
Functional Coverage

As designs become more complex, the only effective way to verify them thoroughly is with constrained-random testing (CRT). This approach elevates you above the tedium of writing individual directed tests, one for each feature in the design. However, if your testbench is taking a random walk through the space of all design states, how do you know if you have reached your destination? Whether you are using random or directed stimulus, you can gauge progress using coverage.

Functional coverage is a measure of which design features have been exercised by the tests. Start with the design specification and create a verification plan with a detailed list of what to test and how. For example, if your design connects to a bus, your tests need to exercise all the possible interactions between the design and bus, including relevant design states, delays, and error modes. The verification plan is a map to show you where to go. For more information on creating a verification plan, see Bergeron (2006).

Use a feedback loop to analyze the coverage results and decide on which actions to take in order to converge on 100% coverage (Figure 9-1). Your first choice is to run existing tests with more seeds; the second is to build new constraints. Resort to creating directed tests only if absolutely necessary.

Figure 9-1 Coverage convergence

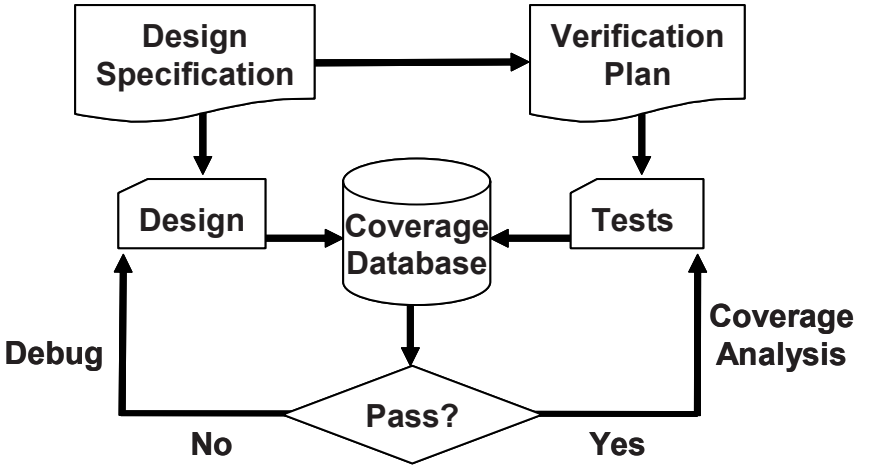


Back when you exclusively wrote directed tests, the verification planning was limited. If the design specification listed 100 features, all you had to do was write 100 tests. Coverage was implicit in the tests – the "register move" test moved all combinations of registers back and forth. Measuring progress was easy: if you had completed 50 tests, you were halfway done. This chapter uses "explicit" and "implicit" to describe how coverage is specified. Explicit coverage is described directly in the test environment using SystemVerilog features. Implicit coverage is implied by a test – when the "register move" directed test passes, you have hopefully covered all register transactions.

With CRT, you are freed from hand crafting every line of input stimulus, but now you need to write code that tracks the effectiveness of the test with respect to the verification plan. You are still more productive, as you are working at a higher level of abstraction. You have moved from tweaking individual bits to describing the interesting design states. Reaching for 100% functional coverage forces you to think more about what you want to observe and how you can direct the design into those states.

Gathering Coverage Data You can run the same random testbench over and over, simply by changing the random seed, to generate new stimulus. Each individual simulation generates a database of functional coverage information, the trail of footprints from the random walk. You can then merge all this information together to measure your overall progress using functional coverage (Figure 9-2).

Figure 9-2 Coverage flow



You then analyze the coverage data to decide how to modify your tests. If the coverage levels are steadily growing, you may just need to run existing tests with new random seeds, or even just run longer tests. If the coverage growth has started to slow, you can add additional constraints to generate more "interesting" stimuli. When you reach a plateau, some parts of the design are not being exercised, and so you need to create more tests. Lastly, when your functional coverage values near 100%, check the bug rate. If bugs are still being found, you may not be measuring true coverage for some areas of your design. Don't be in too big of a rush to reach 100% coverage, which just shows that you looked for bugs in all the usual places. While you are trying to verify your design, take many random walks through the stimulus space; this can create many unanticipated combinations.[1]

Each simulation vendor has its own format for storing coverage data and as well as its own analysis tools. You need to perform the following actions with those tools.

- Run a test with multiple seeds. For a given set of constraints (and coverage groups), compile the testbench and design into a single executeable. Now you need to run this constraint set over and over with different random seeds. You can use the Unix system clock as a seed, but be careful, as your batch system may start multiple jobs simultaneously. These jobs may run on different servers or may start on a single server with multiple processors.
- Check for pass/fail. Functional coverage information is only valid for a successful simulation. When a simulation fails because there is a design bug, the coverage information must be discarded. The coverage data measures how many items in the verification plan are complete, and this plan is based on the design specification. If the design does not match the specification, the coverage values are useless. Some verification teams periodically measure

[1]Thanks to Hans van der Schoot, SJ SNUG, 2007

all functional coverage from scratch so that it reflects the current state of the design.

- Analyze coverage across multiple runs. You need to measure how successful each constraint set is, over time. If you are not yet getting 100% coverage for the areas that are targeted by the constraints, but the amount is still growing, run more seeds. If the coverage level has plateaued, with no recent progress, it is time to modify the constraints. Only if you think that reaching the last few test cases for one particular section may take too long for constrained-random simulation should you consider writing a directed test. Even then, continue to use random stimulus for the other sections of the design, in case this "background noise" finds a bug.

9.1 Coverage Types

Coverage is a generic term for measuring progress to complete design verification. Your simulations slowly paint the canvas of the design, as you try to cover all of the legal combinations. The coverage tools gather information during a simulation and then postprocess it to produce a coverage report. You can use this report to look for coverage holes and then modify existing tests or create new ones to fill the holes. This iterative process continues until you are satisfied with the coverage level.

9.1.1 Code Coverage

The easiest way to measure verification progress is with code coverage. Here you are measuring how many lines of code have been executed (line coverage), which paths through the code and expressions have been executed (path coverage), which single-bit variables have had the values 0 or 1 (toggle coverage), and which states and transitions in a state machine have been visited (FSM coverage). You don't have to write any extra HDL code. The tool instruments your design automatically by analyzing the source code and adding hidden code to gather statistics. You then run all your tests, and the code coverage tool creates a database.

Many simulators include a code coverage tool. A postprocessing tool converts the database into a readable form. The end result is a measure of how much your tests exercise the design code. Note that you are primarily concerned with analyzing the design code, not the testbench. Untested design code could conceal a hardware bug, or may be just redundant code.

Code coverage measures how thoroughly your tests exercised the "implementation" of the design specification, and not the verification plan. Just because your tests have reached 100% code coverage, your job is not done. What if you made a mistake that your test didn't catch? Worse yet, what if your implementation is missing a feature? The following module is for a D-flip flop. Can you see the mistake?

Sample 9.1 Incomplete D-flip flop model missing a path

```
module dff(output logic q, q_l,
           input  logic clk, d, reset_l);

  always @(posedge clk or negedge reset_l) begin
    q <= d;
    q_l <= !d;
  end
endmodule
```

The reset logic was accidently left out. A code coverage tool would report that every line had been exercised, yet the model was not implemented correctly.

9.1.2 Functional Coverage

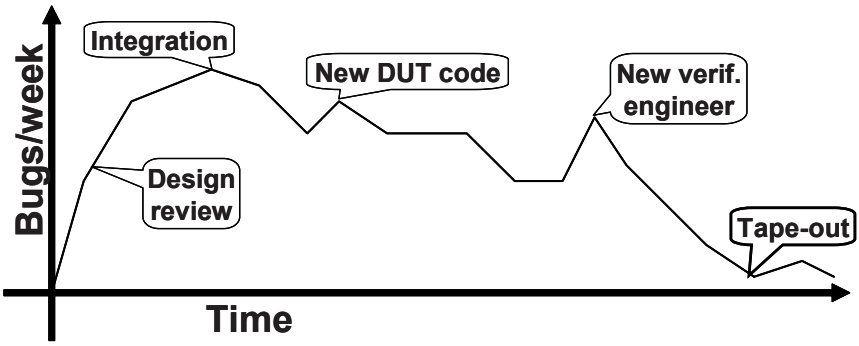
The goal of verification is to ensure that a design behaves correctly in its real environment, be that an MP3 player, network router, or cell phone. The design specification details how the device should operate, whereas the verification plan lists how that functionality is to be stimulated, verified, and measured. When you gather measurements on what functions were covered, you are performing "design" coverage. For example, the verification plan for a D-flip flop would mention not only its data storage but also how it resets to a known state. Until your test checks both these design features, you will not have 100% functional coverage.

Functional coverage is tied to the design intent and is sometimes called "specification coverage," while code coverage measures the design implementation. Consider what happens if a block of code is missing from the design. Code coverage cannot catch this mistake, but functional coverage can.

9.1.3 Bug Rate

An indirect way to measure coverage is to look at the rate at which fresh bugs are found. You should keep track of how many bugs you found each week, over the life of a project. At the start, you may find many bugs through inspection as you create the testbench. As you read the design spec, you may find inconsistencies, which hopefully are fixed before the RTL is written. Once the testbench is up and running, a torrent of bugs is found as you check each module in the system. The bug rate drops, hopefully to zero, as the design nears tape-out. However, you are not yet done. Every time the rate sags, it is time to find different ways to create corner cases.

Figure 9-3 Bug rate during a project



The bug rate can vary per week based on many factors such as project phases, recent design changes, blocks being integrated, personnel changes, and even vacation schedules. Unexpected changes in the rate could signal a potential problem. As shown in Figure 9-3, it is not uncommon to keep finding bugs even after tape-out, and even after the design ships to customers.

9.1.4 Assertion Coverage

Assertions are pieces of declarative code that check the relationships between design signals, either once or over a period of time. These can be simulated along with the design and testbench, or proven by formal tools. Sometimes you can write the equivalent check using SystemVerilog procedural code, but many assertions are more easily expressed using SystemVerilog Assertions (SVA).

Assertions can have local variables and perform simple data checking. If you need to check a more complex protocol, such as determining whether a packet successfully went through a router, procedural code is often better suited for the job. There is a large overlap between sequences that are coded procedurally or using SVA. See Vijayaraghavan and Ramanadhan (2005), Cohen et al. (2005), and Chaps. 3 and 7 in the VMM book, Bergeron et al. (2005) for more information on SVA.

The most familiar assertions look for errors such as two signals that should be mutually exclusive or a request that was never followed by a grant. These error checks should stop the simulation as soon as they detect a problem. Assertions can also check arbitration algorithms, FIFOs, and other hardware. These are coded with the `assert property` statement.

Some assertions might look for interesting signal values or design states, such as a successful bus transaction. These are coded with the `cover property` statement. You can measure how often these assertions are triggered during a test by using assertion coverage. A cover property observes sequences of signals, whereas a cover group (described below) samples data values and transactions during the simulation. These two constructs overlap in that a cover group can trigger when a sequence completes. Additionally, a sequence can collect information that can be used by a cover group.

9.2 Functional Coverage Strategies

Before you write the first line of test code, you need to anticipate what are the key design features, corner cases, and possible failure modes. This is how you write your verification plan. Don't think in terms of data values only; instead, think about what information is encoded in the design. The plan should spell out the significant design states.

9.2.1 Gather Information, Not Data

A classic example is a FIFO. How can you be sure you have thoroughly tested a 1 K FIFO memory? You could measure the values in the read and write indices, but there are over a million possible combinations. Even if you were able to simulate that many cycles, you would not want to read the coverage report.

At a more abstract level, a FIFO can hold from 0 to N–1 possible values. So what if you just compare the read and write indices to measure how full or empty the FIFO is? You would still have 1 K coverage values. If your testbench pushed 100 entries into the FIFO, then pushed in 100 more, do you really need to know if the FIFO ever had 150 values? Not as long as you can successfully read out all values.

The corner cases for a FIFO are Full and Empty. If you can make the FIFO go from Empty (the state after reset) through Full and back down to Empty, you have covered all the levels in between. Other interesting states involve the indices as they pass between all 1's and all 0's. A coverage report for these cases is easy to understand.

You may have noticed that the interesting states are independent of the FIFO size. Once again, look at the information, not the data values.

Design signals with a large range (more than a few dozen possible values) should be broken down into smaller ranges, plus corner cases. For example, your DUT may have a 32-bit address bus, but you certainly don't need to collect 4 billion samples. Check for natural divisions such as memory and IO space. For a counter, pick a few interesting values, and always try to rollover counter values from all 1's back to 0.

9.2.2 Only Measure What You Are Going to Use

Gathering functional coverage data can be expensive, and so only measure what you will analyze and use to improve your tests. Your simulations may run slower as the simulator monitors signals for functional coverage, but this approach has lower overhead than gathering waveform traces and measuring code coverage. Once a simulation completes, the database is saved to disk. With multiple testcases and multiple seeds, you can fill disk drives with functional coverage data and reports. But if you never look at the final coverage reports, don't perform the initial measurements.

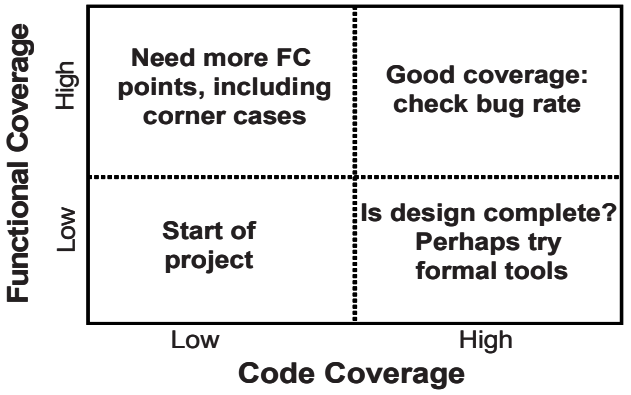
There are several ways to control cover data: at compilation, instantiation, or triggering. You could use switches provided by the simulation vendor, conditional compilation, or suppression of the gathering of coverage data. The last of these is less desirable because the postprocessing report is filled with sections with 0% coverage, making it harder to find the few enabled ones.

9.2.3 Measuring Completeness

Like your kids in the backseat on a family vacation, your manager constantly asks you, "Are we there yet?" How can you tell if you have fully tested a design? You need to look at all coverage measurements and consider the bug rate to see if you have reached your destination.

At the start of a project, both code and functional coverage are low. As you develop tests, run them over and over with different random seeds until you no longer see increasing values of functional coverage. Create additional constraints and tests to explore new areas. Save test/seed combinations that give high coverage, so that you can use them in regression testing (Figure 9-4).

Figure 9-4 Coverage comparison



What if the functional coverage is high but the code coverage is low? Your tests are not exercising the full design, perhaps from an inadequate verification plan. It may be time to go back to the hardware specifications and update your verification plan. Then you need to add more functional coverage points to locate untested functionality.

A more difficult situation is high code coverage but low functional coverage. Even though your testbench is giving the design a good workout, you are unable to put it in all the interesting states. First, see if the design implements all the specified functionality. If the functionality is there, but your tests can't reach it, you might need a formal verification tool that can extract the design's states and create appropriate stimulus.

The goal is both high code and functional coverage. However, don't plan your vacation yet. What is the trend of the bug rate? Are significant bugs still popping up? Worse yet, are they being found deliberately, or did your testbench happen to stumble across a particular combination of states that no one had anticipated? On the other hand, a low bug rate may mean that your existing strategies have run out of steam, and you should look into different approaches. Try different approaches such as new combinations of design blocks and error generators.

9.3 Simple Functional Coverage Example

To measure functional coverage, you begin with the verification plan and write an executable version of it for simulation. In your SystemVerilog testbench, sample the values of variables and expressions. These sample locations are known as cover points. Multiple cover points that are sampled at the same time (such as when a transaction completes) are placed together in a cover group.

Sample 9.2 has a transaction that comes in eight flavors. The testbench generates the `port` variable randomly, and the verification plan requires that every value be tried.

Sample 9.2 Functional coverage of a simple object

```
program automatic test(busifc.TB ifc);

  class Transaction;
    rand bit [31:0] data;
    rand bit [ 2:0] port;         // Eight port numbers
  endclass

  covergroup CovPort;
    coverpoint tr.port;           // Measure coverage
  endgroup

  initial begin
    Transaction tr;
    CovPort ck;
    ck = new();                   // Instantiate group
    tr = new();
    repeat (32) begin             // Run a few cycles
      assert(tr.randomize);       // Create a transaction
      ifc.cb.port <= tr.port;     //   and transmit
      ifc.cb.data <= tr.data;     //   onto interface
      ck.sample();                // Gather coverage
      @ifc.cb;                    // Wait a cycle
    end
  end
endprogram
```

Sample 9.2 creates a random transaction and drives it out to an interface. The testbench samples the value of the `port` field using the `CovPort` cover group. Eight possible values, 32 random transactions – did your testbench generate them all? Here is part of a coverage report from VCS.

Sample 9.3 Coverage report for a simple object

```
Coverpoint Coverage report
CoverageGroup: CovPort
    Coverpoint: tr.port
Summary
    Coverage: 87.50
    Goal: 100
    Number of Expected auto-bins: 8
    Number of User Defined Bins: 0
    Number of Automatically Generated Bins: 7
    Number of User Defined Transitions: 0

    Automatically Generated Bins

    Bin            # hits    at least
    ================================
    auto[1]        7         1
    auto[2]        7         1
    auto[3]        1         1
    auto[4]        5         1
    auto[5]        4         1
    auto[6]        2         1
    auto[7]        6         1
    ================================
```

As you can see, the testbench generated the values 1, 2, 3, 4, 5, 6, and 7, but never generated a `port` of 0. The `at least` column specifies how many hits are needed before a bin is considered covered. See Section 9.9.3 for the `at_least` option.



To improve your functional coverage, the easiest strategy is to just run more simulation cycles, or to try new random seeds. Look at the coverage report for items with two or more hits. Chances are that you just need to make the simulation run longer or to try new seed values. If a cover point had zero or one hit, you probably have to try a new strategy, as the testbench is not creating the proper stimulus. For this example, the very next random transaction (#33) has a `port` value of 0, giving 100% coverage.

Sample 9.4 Coverage report for a simple object, 100% coverage

```
Coverpoint Coverage report
CoverageGroup: CovPort
    Coverpoint: tr.port
Summary
    Coverage: 100
    Goal: 100
    Number of Expected auto-bins: 8
    Number of User Defined Bins: 0
    Number of Automatically Generated Bins: 8
    Number of User Defined Transitions: 0

    Automatically Generated Bins

    Bin           # hits    at least
    ================================
    auto[0]       1         1
    auto[1]       7         1
    auto[2]       7         1
    auto[3]       1         1
    auto[4]       5         1
    auto[5]       4         1
    auto[6]       2         1
    auto[7]       6         1
    ================================
```

9.4 Anatomy of a Cover Group

A cover group is similar to a class – you define it once and then instantiate it one or more times. It contains cover points, options, formal arguments, and an optional trigger. A cover group encompasses one or more data points, all of which are sampled at the same time.

You should create very clear cover group names that explicitly indicate what you are measuring and, if possible, reference to the verification plan. The name `Parity_Errors_In_Hexaword_Cache_Fills` may seem verbose, but when you are trying to read a coverage report that has dozens of cover groups, you will appreciate the extra detail. You can also use the comment option for additional descriptive information, as shown in Section 9.9.2.

A cover group can be defined in a class or at the program or module level. It can sample any visible variable such as program/module variables, signals from an interface, or any signal in the design (using a hierarchical reference). A cover group inside a class can sample variables in that class, as well as data values from embedded classes.



Don't define the cover group in a data class, such as a transaction, as doing so can cause additional overhead when gathering coverage data. Imagine you are trying to track how many beers were consumed by patrons in a pub. Would you try to follow every bottle as it flowed from the loading dock, over the bar, and into each person? No, instead you could just have each patron check off the type and number of beers consumed, as shown in van der Schoot and Bergeron (2006).

In SystemVerilog, you should define cover groups at the appropriate level of abstraction. This level can be at the boundary between your testbench and the design, in the transactors that read and write data, in the environment configuration class, or wherever is needed. The sampling of any transaction must wait until it is actually received by the DUT. If you inject an error in the middle of a transaction, causing it to be aborted in transmission, you need to change how you treat it for functional coverage. You need to use a different cover point that has been created just for error handling.

A class can contain multiple cover groups. This approach allows you to have separate groups that can be enabled and disabled as needed. Additionally, each group may have a separate trigger, allowing you to gather data from many sources.



A cover group must be instantiated for it to collect data. If you forget, no error message about null handles is printed at run-time, but the coverage report will not contain any mention of the cover group. This rule applies for cover groups defined either inside or outside of classes.

9.4.1 Defining a Cover Group in a Class

A cover group can be defined in a program, module, or class. In all cases, you must explicitly instantiate it to start sampling. If the cover group is defined in a class, you do not make a separate name when you instance it; you just use the original cover group name.

Sample 9.5 is very similar to the first example of this chapter except that it embeds a cover group in a transactor class, and thus does not need a separate instance name.

Sample 9.5 Functional coverage inside a class

```
class Transactor;
  Transaction tr;
  mailbox mbx_in;
  covergroup CovPort;
    coverpoint tr.port;
  endgroup

  function new(mailbox mbx_in);
    CovPort = new();          // Instantiate covergroup
    this.mbx_in = mbx_in;
  endfunction

  task main;
    forever begin
      tr = mbx_in.get;        // Get next transaction
      ifc.cb.port <= tr.port; // Send into DUT
      ifc.cb.data <= tr.data;
      CovPort.sample();       // Gather coverage
    end
  endtask

endclass
```

9.5 Triggering a Cover Group

The two major parts of functional coverage are the sampled data values and the time when they are sampled. When new values are ready (such as when a transaction has completed), your testbench triggers the cover group. This can be done directly with the `sample` function, as shown in Sample 9.5, or by using a blocking expression in the `covergroup` definition. The blocking expression can use a `wait` or `@` to block on signals or events.

Use `sample` if you want to explicitly trigger coverage from procedural code, if there is no existing signal or event that tells when to sample, or if there are multiple instances of a cover group that trigger separately.

Use the blocking statement in the `covergroup` declaration if you want to tap into existing events or signals to trigger coverage.

9.5.1 Sampling Using a Callback

One of the better ways to integrate functional coverage into your testbench is to use callbacks, as originally shown in Section 8.7. This technique allows you to build a flexible testbench without restricting when coverage is collected. You can decide for

every point in the verification plan where and when values are sampled. And if you need an extra "hook" in the environment for a callback, you can always add one in an unobtrusive manner, as a callback only "fires" when the test registers a callback object. You can create many separate callbacks for each cover group, with little overhead. As explained in Section 8.7.4, callbacks are superior to using a mailbox to connect the testbench to the coverage objects. You might need multiple mailboxes to collect transactions from different points in your testbench. A mailbox requires a transactor to receive transactions, and multiple mailboxes cause you to juggle multiple threads. Instead of an active transactor, use a passive callback.

Sample 8.30 shows a driver class that has two callback points, before and after the transaction is transmitted. Sample 8.29 shows the base callback class, and Sample 8.31 has a test with an extended callback class that sends data to a scoreboard. Make your own extension, `Driver_cbs_coverage`, of the base callback class, `Driver_cbs`, to call the `sample` task for your cover group in `post_tx`. Push an instance of the coverage callback class into the driver's callback queue, and your coverage code triggers the cover group at the right time. The following two examples define and use the callback `Driver_cbs_coverage`.

Sample 9.6 Test using functional coverage callback

```
program automatic test;
  Environment env;

  initial begin
    Driver_cbs_coverage dcc;

    env = new();
    env.gen_cfg();
    env.build();

    // Create and register the coverage callback
    dcc = new();
    env.drv.cbs.push_back(dcc); // Put into driverÕs Q

    env.run();
    env.wrap_up();
  end

endprogram
```

Sample 9.7 Callback for functional coverage

```
class Driver_cbs_coverage extends Driver_cbs;
  covergroup CovPort;
    ...
  endgroup

  virtual task post_tx(Transaction tr);
    CovPort.sample();          // Sample coverage values
  endtask
endclass
```

9.5.2 Cover Group With an Event Trigger

In Sample 9.8, the cover group `CovPort` is sampled when the testbench triggers the `trans_ready` event.

Sample 9.8 Cover group with a trigger

```
event trans_ready;
covergroup CovPort @(trans_ready);
    coverpoint ifc.cb.port;   // Measure coverage
endgroup
```

The advantage of using an event over calling the `sample` method directly is that you may be able to use an existing event such as one triggered by an assertion, as shown in Sample 9.10.

9.5.3 Triggering on a SystemVerilog Assertion

If you already have an SVA that looks for useful events like a complete transaction, you can add an event trigger to wake up the cover group.

Sample 9.9 Module with SystemVerilog Assertion

```
module mem(simple_bus sb);
  bit [7:0] data, addr;
  event write_event;

  cover property
   (@(posedge sb.clock) sb.write_ena==1)
    -> write_event;
endmodule
```

Sample 9.10 Triggering a cover group with an SVA

```
program automatic test(simple_bus sb);

  covergroup Write_cg @($root.top.m1.write_event);
    coverpoint $root.top.m1.data;
    coverpoint $root.top.m1.addr;
  endgroup

  Write_cg wcg;

  initial begin
    wcg = new();
    // Apply stimulus here
    sb.write_ena <= 1;
    ...
    #10000 $finish;
  end
endprogram
```

9.6 Data Sampling

How is coverage information gathered? When you specify a variable or expression in a cover point, SystemVerilog creates a number of "bins" to record how many times each value has been seen. These bins are the basic units of measurement for functional coverage. If you sample a one-bit variable, a maximum of two bins are created. You can imagine that SystemVerilog drops a token in one or the other bin every time the cover group is triggered. At the end of each simulation, a database is created with all bins that have a token in them. You then run an analysis tool that reads all databases and generates a report with the coverage for each part of the design and for the total coverage.

9.6.1 Individual Bins and Total Coverage

To calculate the coverage for a point, you first have to determine the total number of possible values, also known as the domain. There may be one value per bin or multiple values. Coverage is the number of sampled values divided by the number of bins in the domain.

A cover point that is a 3-bit variable has the domain 0:7 and is normally divided into eight bins. If, during simulation, values belonging to seven bins are sampled, the report will show 7/8 or 87.5% coverage for this point. All these points are combined to show the coverage for the entire group, and then all the groups are combined to give a coverage percentage for all the simulation databases.

This is the status for a single simulation. You need to track coverage over time. Look for trends so that you can see where to run more simulations or add new constraints or tests. Now you can better predict when verification of the design will be completed.

9.6.2 Creating Bins Automatically

As you saw in the report in Sample 9.3, SystemVerilog automatically creates bins for cover points. It looks at the domain of the sampled expression to determine the range of possible values. For an expression that is N bits wide, there are 2^N possible values. For the 3-bit variable `port`, there are 8 possible values. The range of an enumerated type is shown in Section 9.6.8. The domain for enumerated data types is the number of named values. You can also explicitly define bins as shown in Section 9.6.5.

9.6.3 Limiting the Number of Automatic Bins Created

The cover group option `auto_bin_max` specifies the maximum number of bins to automatically create, with a default of 64 bins. If the domain of values in the cover point variable or expression is greater than this option, SystemVerilog divides the range into `auto_bin_max` bins. For example, a 16-bit variable has 65,536 possible values, and so each of the 64 bins covers 1,024 values.

In reality, you may find this approach impractical, as it is very difficult to find the needle of missing coverage in a haystack of auto-generated bins. Lowering this limit to 8 or 16, or better yet, explicitly define the bins as shown in Section 9.6.5.

The following code takes the chapter's first example and adds a cover point option that sets `auto_bin_max` to two bins. The sampled variable is still `port`, which is three bits wide, for a domain of eight possible values. The first bin holds the lower half of the range, 0–3, and the other hold the upper values, 4–7.

Sample 9.11 Using `auto_bin_max` set to 2

```
covergroup CovPort;
  coverpoint tr.port
    { options.auto_bin_max = 2; }  // Divide into 2 bins
endgroup
```

The coverage report from VCS shows the two bins. This simulation achieved 100% coverage because the eight `port` values were mapped to two bins. Since both bins have sampled values, your coverage is 100%.

Sample 9.12 Report with `auto_bin_max` set to 2

```
Bin              # hits    at least
================================
auto[0:3]        15        1
auto[4:7]        17        1
```

Sample 9.11 used `auto_bin_max` as an option for the cover point only. You can also use it as an option for the entire group.

Sample 9.13 Using `auto_bin_max` for all cover points

```
covergroup CovPort;
  options.auto_bin_max = 2; // Affects port & data
  coverpoint tr.port;
  coverpoint tr.data;
endgroup
```

9.6.4 Sampling Expressions

You can sample expressions, but always check the coverage report to be sure you are getting the values you expect. You may have to adjust the width of the computed expression, as shown in Section 2.15. For example, sampling a 3-bit header length (0:7) plus a 4-bit payload length (0:15) creates only 2^4 or 16 bins, which may not be enough if your transactions can actually be 0:23 bytes long.

Sample 9.14 Using an expression in a cover point

```
class Transaction;
  rand bit [2:0] hdr_len;      // range: 0:7
  rand bit [3:0] payload_len;  // range: 0:15
  rand bit [3:0] kind;         // range: 0:15
  ...
endclass

Transaction tr;

covergroup CovLen;
  len16: coverpoint (tr.hdr_len + tr.payload_len);
  len32: coverpoint (tr.hdr_len + tr.payload_len + 5'b0);
endgroup
```

Sample 9.14 has a cover group that samples the total transaction length. The cover point has a label to make it easier to read the coverage report. Also, the expression has an additional dummy constant so that the transaction length is computed with 5-bit precision, for a maximum of 32 auto-generated bins.

A quick run with 200 transactions showed that the `len16` had 100% coverage, but this is across only 16 bins. The cover point `len32` had 68% coverage across 32 bins. Neither of these cover points are correct, as the maximum length has a domain of 0:22 (0+0:7+15). The auto-generated bins just don't work, as the maximum length is not a power of 2.

9.6.5 User-Defined Bins Find a Bug

Automatically generated bins are okay for anonymous data values, such as counter values, addresses, or values that are a power of 2. For other values, you should explicitly name the bins to improve accuracy and ease coverage report analysis. SystemVerilog automatically creates bin names for enumerated types, but for other variables you need to give names to the interesting states. The easiest way to specify bins is with the `[]` syntax, as shown in Sample 9.15.

Sample 9.15 Defining bins for transaction length

```
covergroup CovLen;
  len: coverpoint (tr.hdr_len + tr.payload_len + 5'b0)
    {bins len[] = {[0:23]}; }
endgroup
```

After sampling 2,000 random transactions, the group has 95.83% coverage. A quick look at the report shows the problem – the length of 23 (17 hex) was never seen. The longest header is 7, and the longest payload is 15, for a total of 22, not 23! If you change to the `bins` declaration to use 0:22, the coverage jumps to 100%. The user-defined bins found a bug in the test.

Sample 9.16 Coverage report for transaction length

```
Bin         # hits    at least
============================
len_00      13        1
len_01      36        1
len_02      51        1
len_03      60        1
len_04      72        1
len_05      88        1
len_06      127       1
len_07      122       1
len_08      133       1
len_09      138       1
len_0a      115       1
len_0b      128       1
len_0c      125       1
len_0d      111       1
len_0e      115       1
len_0f      134       1
len_10      107       1
len_11      102       1
len_12      70        1
len_13      65        1
len_14      39        1
len_15      30        1
len_16      19        1
```

```
len_17     0          1
============================
```

9.6.6 Naming the Cover Point Bins

Sample 9.17 samples a 4-bit variable, `kind`, that has 16 possible values. The first bin is called `zero` and counts the number of times that `kind` is 0 when sampled. The next four values, 1–3 and 5, are all grouped into a single bin, `lo`. The upper eight values, 8–15, are kept in separate bins, `hi_8`, `hi_9`, `hi_a`, `hi_b`, `hi_c`, `hi_d`, `hi_e`, and `hi_f`. Note how `$` in the `hi` bin expression is used as a shorthand notation for the largest value for the sampled variable. Lastly, `misc` holds all values that were not previously chosen: 4, 6, and 7.

Sample 9.17 Specifying bin names

```
covergroup CovKind;
  coverpoint tr.kind {
    bins zero = {0};          // 1 bin for kind==0
    bins lo   = {[1:3], 5};   // 1 bin for values 1:3, 5
    bins hi[] = {[8:$]};      // 8 separate bins: 8...15
    bins misc = default;      // 1 bin for all the rest
  }                           // No semicolon
endgroup // CoverKind
```

Note that the additional information about the `coverpoint` is grouped using curly braces: `{}`. This is because the bin specification is declarative code, not procedural code that would be grouped with `begin...end`. Lastly, the final curly brace is NOT followed by a semicolon, just as an `end` never is.

Now you can easily see which bins have no hits – `hi_8` in this case.

Sample 9.18 Report showing bin names

```
Bin         # hits    at least
============================
hi_8         0         1
hi_9         5         1
hi_a         3         1
hi_b         4         1
hi_c         2         1
hi_d         2         1
hi_e         9         1
hi_f         4         1
lo          16         1
misc        15         1
zero         1         1
```

When you define the bins, you are restricting the values used for coverage to those that are interesting to you. SystemVerilog no longer automatically creates bins, and it ignores values that do not fall into a predefined bin. More importantly, only the bins you create are used to calculate functional coverage. You get 100% coverage only as long as you get a hit in every specified bin.



Values that do not fall into any specified bin are ignored. This rule is useful if the sampled value, such as transaction length, is not a power of 2. In general, if you are specifying bins, always use the `default` bin specifier to catch values that you may have forgotten.

In Sample 9.17, the range for `hi` uses a dollar sign (`$`) on the right side to specify the upper value. This is a very useful shortcut – now you can let the compiler calculate the limits for a range. You can use the dollar sign on the left side of a range to specify the lower limit. In Sample 9.19, the `$` in the range for bin `neg` represents the negative number furthest from zero: 32'h8000_0000, or –2,147,483,648, whereas the `$` in bin `pos` represents the largest signed positive value, 32'h7FFF_FFFF, or 2,147,483,647.

Sample 9.19 Specifying ranges with $

```
int i;
covergroup range_cover;
  coverpoint i {
    bins neg  = {[$:-1]};  // Negative values
    bins zero = {0};       // Zero
    bins pos  = {[1:$]};   // Positive values
  }
endgroup
```

9.6.7 Conditional Coverage

You can use the `iff` keyword to add a condition to a cover point. The most common reason for doing so is to turn off coverage during reset so that stray triggers are ignored. Sample 9.20 gathers only values of `port` when `reset` is 0, where `reset` is active-high.

Sample 9.20 Conditional coverage – disable during reset

```
covergroup CoverPort;
  // DonÕt gather coverage when reset==1
  coverpoint port iff (!bus_if.reset);
endgroup
```

Alternately, you can use the `start` and `stop` functions to control individual instances of cover groups.

Sample 9.21 Using `stop` and `start` functions

```
initial begin
  CovPort ck = new();       // Instantiate cover group

  // Reset sequence stops collection of coverage data
  #1ns ck.stop();
  bus_if.reset = 1;

  #100ns bus_if.reset = 0; // End of reset
  ck.start();
  ...
end
```

9.6.8 Creating Bins for Enumerated Types

For enumerated types, SystemVerilog creates a bin for each value.

Sample 9.22 Functional coverage for an enumerated type

```
typedef enum {INIT, DECODE, IDLE} fsmstate_e;
fsmstate_e pstate, nstate;   // declare typed variables
covergroup cg_fsm;
  coverpoint pstate;
endgroup
```

Here is part of the coverage report from VCS, showing the bins for the enumerated types.

Sample 9.23 Coverage report with enumerated types

```
Bin              # hits    at least
================================
auto_DECODE      11        1
auto_IDLE        11        1
auto_INIT        10        1
================================
```

If you want to group multiple values into a single bin, you have to define your own bins. Any bins outside the enumerated values are ignored unless you define a bin with the default specifier. When you gather coverage on enumerated types, `auto_bin_max` does not apply.

9.6.9 Transition Coverage

You can specify state transitions for a cover point. In this way, you can tell not only what interesting values were seen but also the sequences. For example, you can check if `port` ever went from 0 to 1, 2, or 3.

Sample 9.24 Specifying transitions for a cover point

```
covergroup CoverPort;
  coverpoint port {
    bins t1 = (0 => 1), (0 => 2), (0 => 3);
  }
endgroup
```

You can quickly specify multiple transitions using ranges. The expression `(1,2 => 3,4)` creates the four transitions `(1=>3)`, `(1=>4)`, `(2=>3)`, and `(2=>4)`.

You can specify transitions of any length. Note that you have to sample once for each state in the transition. So `(0 => 1 => 2)` is different from `(0 => 1 => 1 => 2)` or `(0 => 1 => 1 => 1 => 2)`. If you need to repeat values, as in the last sequence, you can use the shorthand form: `(0 => 1[*3] => 2)`. To repeat the value 1 for 3, 4, or 5 times, use `1[*3:5]`.

9.6.10 Wildcard States and Transitions

You use the `wildcard` keyword to create multiple states and transitions. Any `X`, `Z`, or `?` in the expression is treated as a wildcard for 0 or 1. The following creates a cover point with a bin for even values and one for odd.

Sample 9.25 Wildcard bins for a cover point

```
bit [2:0] port;
covergroup CoverPort;
  coverpoint port {
    wildcard bins even = {3'b??0};
    wildcard bins odd  = {3'b??1};
  }
endgroup
```

9.6.11 Ignoring Values

With some cover points, you never get all possible values. For instance, a 3-bit variable may be used to store just six values, 0–5. If you use automatic bin creation, you never get beyond 75% coverage. There are two ways to solve this problem. You can explicitly define the bins that you want to cover as shows in Section 9.6.5. Alternatively, you can let SystemVerilog automatically create bins, and then use `ignore_bins` to tell which values to exclude from functional coverage calculation.

Sample 9.26 Cover point with `ignore_bins`

```
bit [2:0] low_ports_0_5;      // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    ignore_bins hi = {[6,7]}; // Ignore upper 2 bins
  }
endgroup
```

The original range of `low_ports_0_5`, a three-bit variable is 0:7. The `ignore_bins` excludes the last two bins, which reduces the range to 0:5. So total coverage for this group is the number of bins with samples, divided by the total number of bins, which is 5 in this case.

Sample 9.27 Cover point with `auto_bin_max` and `ignore_bins`

```
bit [2:0] low_ports_0_5;      // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    options.auto_bin_max = 4; // 0:1, 2:3, 4:5, 6:7
    ignore_bins hi = {[6,7]}; // Ignore upper 2 values
  }
endgroup
```

If you define bins either explicitly or by using the `auto_bin_max` option, and then ignore them, the ignored bins do not contribute to the calculation of coverage. In Sample 9.27, four bins are initially created using the `auto_bin_max` option: 0:1, 2:3, 4:5, and 6:7. However, then the uppermost bin is eliminated by `ignore_bins`, and so in the end only three bins are created. This cover point can have coverage of 0%, 33%, 66%, or 100%

9.6.12 Illegal Bins

Some sampled values not only should be ignored but also should cause an error if they are seen. This is best done in the testbench's monitor code, but can also be done by labeling a bin with `illegal_bins`. Use `illegal_bins` to catch states that were missed by the test's error checking. This also double-checks the accuracy of your bin creation: if an illegal value is found by the cover group, it is a problem either with the testbench or with your bin definitions.

Sample 9.28 Cover point with `illegal_bins`

```
bit [2:0] low_ports_0_5;      // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    illegal_bins hi = {[6,7]}; // Give error if seen
  }
endgroup
```

9.6.13 State Machine Coverage

You should have noticed that if a cover group is used on a state machine, you can use bins to list the specific states, and transitions for the arcs. However, this does not mean you should use SystemVerilog's functional coverage to measure state machine coverage. You would have to extract the states and arcs manually. Even if you did this correctly the first time, you might miss future changes to the design code. Instead, use a code coverage tool that extracts the state register, states, and arcs automatically, saving you from possible mistakes.

However, an automatic tool extracts the information exactly as coded, mistakes and all. You may want to monitor small, critical state machines manually using functional coverage.

9.7 Cross Coverage

A cover point records the observed values of a single variable or expression. You may want to know not only what bus transactions occurred but also what errors happened during those transactions, and their source and destination. For this you need cross coverage that measures what values were seen for two or more cover points at the same time. Note that when you measure cross coverage of a variable with `N` values, and of another with `M` values, SystemVerilog needs `NxM` cross bins to store all the combinations.

9.7.1 Basic Cross Coverage Example

Previous examples have measured coverage of the transaction kind, and port number, but what about the two combined? Did you try every kind of transaction into every port? The `cross` construct in SystemVerilog records the combined values of two or more cover points in a group. The `cross` statement takes only cover points or a simple variable name. If you want to use expressions, hierarchical names or variables in an object such as `handle.variable`, you must first specify the expression in a `coverpoint` with a label and then use the label in the `cross` statement.

Sample 9.29 creates cover points for `tr.kind` and `tr.port`. Then the two points are crossed to show all combinations. SystemVerilog creates a total of 128 (8×16) bins. Even a simple cross can result in a very large number of bins.

Sample 9.29 Basic cross coverage

```
class Transaction;
  rand bit [3:0] kind;
  rand bit [2:0] port;
endclass

Transaction tr;

covergroup CovPort;
  kind: coverpoint tr.kind;  // Create cover point kind
  port: coverpoint tr.port;  // Create cover point port
  cross kind, port;          // Cross kind and port
endgroup
```

A random testbench created 200 transactions and produced the coverage report in Sample 9.30. Note that even though all possible `kind` and `port` values were generated, about 1/8 of the cross combinations were not seen.

Sample 9.30 Coverage summary report for basic cross coverage

```
Cumulative report for Transaction::CovPort
Summary:
    Coverage: 95.83
    Goal: 100

Coverpoint          Coverage     Goal         Weight
=======================================================
kind                  100.00      100         1
port                  100.00      100         1
=======================================================
Cross                 Coverage    Goal         Weight
=======================================================
Transaction::CovPort  87.50       100         1

Cross Coverage report
CoverageGroup: Transaction::CovPort
    Cross: Transaction::CovPort
Summary
    Coverage: 87.50
    Goal: 100
    Coverpoints Crossed: kind port
    Number of Expected Cross Bins: 128
    Number of User Defined Cross Bins: 0
    Number of Automatically Generated Cross Bins: 112
```

```
    Automatically Generated Cross Bins

    kind       port           # hits       at least
 ================================================
    auto[0]    auto[0]        1            1
    auto[0]    auto[1]        4            1
    auto[0]    auto[2]        3            1
    auto[0]    auto[5]        1            1
...
```

9.7.2 Labeling Cross Coverage Bins

If you want more readable cross coverage bin names, you can label the individual cover point bins, and SystemVerilog will use these names when creating the cross bins.

Sample 9.31 Specifying cross coverage bin names

```
covergroup CovPortKind;
  port: coverpoint tr.port
   {bins port[] = {[0:$]};
   }
  kind: coverpoint tr.kind
   {bins zero = {0};       // 1 bin for kind==0
    bins lo   = {[1:3]};   // 1 bin for values 1:3
    bins hi[] = {[8:$]};   // 8 separate bins
    bins misc = default;   // 1 bin for all the rest
  }
  cross kind, port;
endgroup
```

If you define bins that contain multiple values, the coverage statistics change. In the report below, the number of bins has dropped from 128 to 88. This is because `kind` has 11 bins: `zero`, `lo`, `hi_8`, `hi_9`, `hi_a`, `hi_b`, `hi_c`, `hi_d`, `hi_e`, `hi_f`, and `misc`. The percentage of coverage jumped from 87.5% to 90.91% because any single value in the `lo` bin, such as 2, allows that bin to be marked as covered, even if the other values, 0 or 3, are not seen.

Sample 9.32 Cross coverage report with labeled bins

```
Summary
    Coverage: 90.91
    Number of Coverpoints Crossed: 2
    Coverpoints Crossed: kind port
    Number of Expected Cross Bins: 88
    Number of Automatically Generated Cross Bins: 80
    Automatically Generated Cross Bins

    port          kind           # hits       at least
 ================================================
    port_0        hi_8           3            1
    port_0        hi_a           1            1
    port_0        hi_b           4            1
    port_0        hi_c           4            1
    port_0        hi_d           4            1
    port_0        hi_e           1            1
    port_0        lo             7            1
    port_0        misc           6            1
    port_0        zero           1            1
    port_1        hi_8           3            1
...
```

9.7.3 Excluding Cross Coverage Bins

To reduce the number of bins, use `ignore_bins`. With cross coverage, you specify the cover point with `binsof` and the set of values with `intersect` so that a single `ignore_bins` construct can sweep out many individual bins.

Sample 9.33 Excluding bins from cross coverage

```
covergroup Covport;
  port: coverpoint tr.port
   {bins port[] = {[0:$]};
   }
  kind: coverpoint tr.kind {
    bins zero = {0};       // 1 bin for kind==0
    bins lo   = {[1:3]};   // 1 bin for values 1:3
    bins hi[] = {[8:$]};   // 8 separate bins
    bins misc = default;   // 1 bin for all the rest
  }
  cross kind, port {
    ignore_bins hi = binsof(port) intersect {7};
    ignore_bins md = binsof(port) intersect {0} &&
                     binsof(kind) intersect {[9:11]};
    ignore_bins lo = binsof(kind.lo);
  }
endgroup
```

The first `ignore_bins` just excludes bins where `port` is 7 and any value of `kind`. Since `kind` is a 4-bit value, this statement excludes 16 bins. The second `ignore_bins` is more selective, ignoring bins where `port` is 0 and `kind` is 9, 10, or 11, for a total of 3 bins.

The `ignore_bins` can use the bins defined in the individual cover points. The `ignore_bins lo` uses bin names to exclude `kind.lo` that is 1, 2, or 3. The bins must be names defined at compile-time, such as `zero` and `lo`. The bins `hi_8`, `hi_9`, `hi_a`,... `hi_f`, and any automatically generated bins do not have names that can be used at compile-time in other statements such as `ignore_bins`; these names are created at run-time or during the report generation.

Note that `binsof` uses parentheses `()`, while `intersect` specifies a range and therefore uses curly braces `{}`.

9.7.4 Excluding Cover Points From the Total Coverage Metric

The total coverage for a group is based on all simple cover points and cross coverage. If you are only sampling a variable or expression in a `coverpoint` to be used in a `cross` statement, you should set its weight to 0 so that it does not contribute to the total coverage.

Sample 9.34 Specifying cross coverage weight

```
covergroup CovPort;
  kind: coverpoint tr.kind
   {bins zero = {0};
    bins lo   = {[1:3]};
    bins hi[] = {[8:$]};
    bins misc = default;
    option.weight = 5;      // Count in total
  }
  port: coverpoint tr.port
   {bins port[] = {[0:$]};
    option.weight = 0;      // DonÕt count towards total
  }
  cross kind, port
    {option.weight = 10;}  // Give cross extra weight
endgroup
```

9.7.5 Merging Data From Multiple Domains

One problem with cross coverage is that you may need to sample values from different timing domains. You might want to know if your processor ever received an interrupt in the middle of a cache fill. The interrupt hardware is separate from and may use different clocks than the cache hardware, making it difficult to know when to

trigger the cover group. On the other hand, you want to make sure you have tested this case, as a previous design had a bug of this very sort.

The solution is to create a timing domain separate from the cache or interrupt hardware. Make copies of the signals into temporary variables and then sample them in a new coverage group that measures the cross coverage.

9.7.6 Cross Coverage Alternatives

As your cross coverage definition becomes more elaborate, you may spend considerable time specifying which bins should be used and which should be ignored. You may have two random bits, `a` and `b` with three interesting states, {`a==0`, `b==0`}, {`a==1`, `b==0`}, and {`b==1`}.

Sample 9.35 shows how you can name bins in the cover points and then gather cross coverage using those bins.

Sample 9.35 Cross coverage with bin names

```
class Transaction;
  rand bit a, b;
endclass

covergroup CrossBinNames;
  a: coverpoint tr.a
     { bins a0 = {0};
       bins a1 = {1};
       option.weight=0;}    // DonÕt count this coverpoint
  b: coverpoint tr.b
     { bins b0 = {0};
       bins b1 = {1};
       option.weight=0;}    // DonÕt count this coverpoint
  ab: cross a, b
     { bins a0b0 = binsof(a.a0) && binsof(b.b0);
       bins a1b0 = binsof(a.a1) && binsof(b.b0);
       bins b1   = binsof(b.b1); }
endgroup
```

Sample 9.36 gathers the same cross coverage, but now uses `binsof` to specify the cross coverage values.

Sample 9.36 Cross coverage with `binsof`

```
class Transaction;
  rand bit a, b;
endclass

covergroup CrossBinsofIntersect;
  a: coverpoint tr.a
     { option.weight=0; }   // DonÕt count this coverpoint
  b: coverpoint tr.b
     { option.weight=0; }   // DonÕt count this coverpoint
  ab: cross a, b
       { bins a0b0 = binsof(a) intersect {0} &&
                     binsof(b) intersect {0};
         bins a1b0 = binsof(a) intersect {1} &&
                     binsof(b) intersect {0};
         bins b1   = binsof(b) intersect {1}; }
endgroup
```

Alternatively, you can make a cover point that samples a concatenation of values. Then you only have to define bins using the less complex cover point syntax.

Sample 9.37 Mimicking cross coverage with concatenation

```
covergroup CrossManual;
  ab: coverpoint {tr.a, tr.b}
    { bins a0b0 = {2'b00};
      bins a1b0 = {2'b10};
      wildcard bins b1 = {2'b?1};
    }
endgroup
```

Use the style in Sample 9.35 if you already have bins defined for the individual cover points and want to use them to build the cross coverage bins. Use Sample 9.36 if you need to build cross coverage bins but have no predefined cover point bins. Use Sample 9.37 if you want the tersest format.

9.8 Generic Cover Groups

As you start writing cover groups, you will find that some are very close to one another. SystemVerilog allows you to create a generic cover group so that you can specify a few unique details when you instantiate it. SystemVerilog does not allow you to pass the cover group trigger argument into an instance. As a workaround, you can put a coverage group into a class and pass the trigger into the constructor.

9.8.1 Pass Cover Group Arguments by Value

Sample 9.38 shows a cover group that uses an argument to split the range into two halves. Just pass the midpoint value to the cover groups' `new` function.

Sample 9.38 Simple argument

```
bit [2:0] port;     // Values: 0:7

covergroup CoverPort (int mid);
  coverpoint port
   {bins lo = {[0:mid-1]};
    bins hi = {[mid:$]};
  }
endgroup

CoverPort cp;
initial
  cp = new(5);           // lo=0:4, hi=5:7
```

9.8.2 Pass Cover Group Arguments by Reference

You can specify a variable to be sampled with pass-by-reference. Here you want the cover group to sample the value during the entire simulation, not just to use the value when the constructor is called.

Sample 9.39 Pass-by-reference

```
bit [2:0] port_a, port_b;

covergroup CoverPort (ref bit [2:0] port, input int mid);
  coverpoint port {
    bins lo = {[0:mid-1]};
    bins hi = {[mid:$]};
  }
endgroup

CoverPort cpa, cpb;
initial
  begin
    cpa = new(port_a, 4);  // port_a, lo=0:3, hi=4:7
    cpb = new(port_b, 2);  // port_b, lo=0:1, hi=2:7
  end
```



Like a task or function, the arguments to a cover group have a sticky direction. In Sample 9.39, if you forgot the `input` direction, the `mid` argument will have the direction `ref`. The example would not compile because you cannot pass a constant (4 or 2) into a `ref` argument.[2]

9.9 Coverage Options

You can specify additional information in the cover group using options. There are two flavors of options: instance options that apply to a specific cover group instance and type options that apply to all instances of the cover group, and are analogous to static data members of classes. Options can be placed in the cover group so that they apply to all cover points in the group, or they can be put inside a single cover point for finer control. You have already seen the `auto_bin_max` and `weight` options. Here are several more.

9.9.1 Per-Instance Coverage

If your testbench instantiates a coverage group multiple times, by default SystemVerilog groups together all the coverage data from all the instances. However, if you have several generators, each creating very different streams of transactions, you will need to see separate reports. For example, one generator may be creating long transactions while another makes short ones. The cover group in Sample 9.40 can be instantiated in each separate generator. It keeps track of coverage for each instance, and has a unique comment string with the hierarchical path to the cover group instance.

Sample 9.40 Specifying per-instance coverage

```
covergroup CoverLength;
  coverpoint tr.length;
  option.per_instance = 1;
  // Use hierarchical path in comment
  option.comment = $psprintf("%m");
endgroup
```

The per-instance option can only be given in the cover group, not in the cover point or cross point.

9.9.2 Cover Group Comment

You can add a comment into coverage reports to make them easier to analyze. A comment could be as simple as the section number from the verification plan to tags used

[2]The P1800-2005 LRM has a similar example showing that the direction is NOT sticky, but this is a mistake and will be corrected in the next version. Can you find another mistake in that LRM example?

by a report parser to automatically extract relevant information from the sea of data. If you have a cover group that is only instantiated once, use the `type` option as shown in Sample 9.41.

Sample 9.41 Specifying comments for a cover group

```
covergroup CoverPort;
  type_option.comment = "Section 3.2.14 Port numbers";
  coverpoint port;
endgroup
```

However, if you have multiple instances, you can give each a separate comment, as long as you also use the per-instance option.

Sample 9.42 Specifying comments for a cover group instance

```
covergroup CoverPort(int lo,hi, string comment);
  option.comment = comment;
  option.per_instance = 1;
  coverpoint port
    {bins range = {[lo:hi]};
     }
endgroup
...
CoverPort cp_lo = new(0,3, "Low port numbers");
CoverPort cp_hi = new(4,7, "High port numbers");
```

9.9.3 Coverage Threshold

You may not have sufficient visibility into the design to gather robust coverage information. Suppose you are verifying that a DMA state machine can handle bus errors. You don't have access to its current state, but you know the range of cycles that are needed for a transfer. So if you repeatedly cause errors during that range, you have probably covered all the states. So you could set `option.at_least` to 8 or more to specify that after 8 hits on a bin, you are confident that you have exercised that combination Figure 9-6.

If you define `option.at_least` at the cover group level, it applies to all cover points. If you define it inside a point, it only applies to that single point.

However, as Sample 9.2 showed, even after 32 attempts, the random `kind` variable still did not hit all possible values. So only use `at_least` if there is no direct way to measure coverage.

9.9.4 Printing the Empty Bins

By default, the coverage report shows only the bins with samples. Your job is to verify all that is listed in the verification plan, and so you are actually more interested in

the bins without samples. Use the option `cross_num_print_missing` to tell the simulation and report tools to show you all bins, especially the ones with no hits. Set it to a large value, as shown in Sample 9.43, but no larger than you are willing to read.

Sample 9.43 Report all bins including empty ones

```
covergroup CovPort;
  kind: coverpoint tr.kind;
  port: coverpoint tr.port
  cross kind, port;
  option.cross_num_print_missing = 1_000;
endgroup
```

9.9.5 Coverage Goal

The goal for a cover group or point is the level at which the group or point is considered fully covered. The default is 100% coverage. If you set this level below 100%, you are requesting less than complete coverage, which is probably not desirable. This option affects only the coverage report.

Sample 9.44 Specifying the coverage goal

```
covergroup CoverPort;
  coverpoint port;
  option.goal = 90;  // Settle for partial coverage
endgroup
```

9.10 Analyzing Coverage Data

In general, assume you need more seeds and fewer constraints. After all, it is easier to run more tests than to construct new constraints. If you are not careful, new constraints can easily restrict the search space.

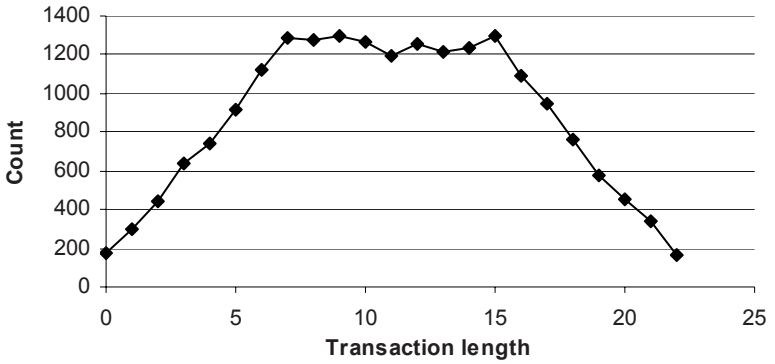
If your cover point has only zero or one sample, your constraints are probably not targeting these areas at all. You need to add constraints that "pull" the solver into new areas. In Sample 9.15, the transaction length had an uneven distribution. This situation is similar to the distribution seen when you roll two dice and look at the total value.

Sample 9.45 Original class for transaction length

```
class Transaction;
  rand bit [2:0] hdr_len;
  rand bit [3:0] payload_len;
  rand bit [4:0] len;
  constraint length {len == hdr_len + payload_len; }
endclass
```

The problem with this class is that `len` is not evenly weighted (Figure 9-5).

Figure 9-5 Uneven probability for transaction length

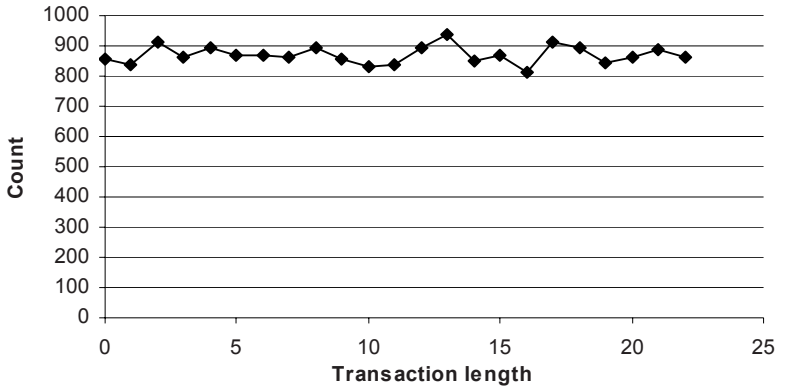


If you want to make the total length be evenly distributed, use a `solve...before` constraint (Figure 9-6).

Sample 9.46 `solve...before` constraint for transaction length

```
constraint length {len == hdr_len + payload_len;
                   solve len before hdr_len, payload_len; }
```

Figure 9-6 Even probability for transaction length with `solve...before`



The normal alternative to `solve...before` is the `dist` constraint. However, this does not work, as `len` is also being constrained by the sum of the two lengths.

9.11 Measuring Coverage Statistics During Simulation

You can query the level of functional coverage on the fly during simulation. This allows you to check whether you have reached your coverage goals, and possibly to control a random test.

At the global level, you can get the total coverage of all cover groups with `$get_coverage`, which returns a real number between 0 and 100. This system task looks across all cover groups.

You can narrow down your measurements with the `get_coverage()` and `get_inst_coverage()` methods. The first function works with both cover group names and instances to give coverage across all instances of a cover group, for example, `CoverGroup::get_coverage()` or `cgInst.get_coverage()`. The second function returns coverage for a specific cover group instance, for example `cgInst.get_inst_coverage()`. You need to specify `option.per_instance=1` if you want to gather perinstance coverage.

The most practical use for these functions is to monitor coverage over a long test. If the coverage level does not advance after a given number of transactions or cycles, the test should stop. Hopefully, another seed or test will increase the coverage.

While it would be nice to have a test that can perform some sophisticated actions based on functional coverage results, it is very hard to write this sort of test. Each test + random seed pair may uncover new functionality, but it may take many runs to reach a goal. If a test finds that it has not reached 100% coverage, what should it do? Run for more cycles? How many more? Should it change the stimulus being generated? How can you correlate a change in the input with the level of functional coverage? The one reliable thing to change is the random seed, which you should only do once per simulation. Otherwise, how can you reproduce a design bug if the stimulus depends on multiple random seeds?

You can query the functional coverage statistics if you want to create your own coverage database. Verification teams have built their own SQL databases that are fed functional coverage data from simulation. This setup allows them greater control over the data, but requires a lot of work outside of creating tests.

Some formal verification tools can extract the state of a design and then create input stimulus to reach all possible states. Don't try to duplicate this in your testbench!

9.12 Conclusion

When you switch from writing directed tests, hand-crafting every bit of stimulus, to CRT, you might worry that the tests are no longer under your command. By measuring coverage, especially functional coverage, you regain control by knowing what features have been tested.

Using functional coverage requires a detailed verification plan and much time creating the cover groups, analyzing the results, and modifying tests to create the proper stimulus. This may seem like a lot of work, but is less effort than would be required to write the equivalent directed tests. Additionally, the time spent in gathering coverage helps you better track your progress in verifying your design.

Chapter 10

Advanced Interfaces

In Chap. 4 you learned how to connect the design and testbench with interfaces. These physical interfaces represent real signals, similar to the wires that connected ports in Verilog-1995. A testbench uses these interfaces by statically connecting to them through ports. However, for many designs, the testbench needs to connect dynamically to the design.

For example, in a network switch, a single driver class may connect to many interfaces, one for each input channel of the DUT. You wouldn't want to write a unique driver for each channel – instead you want to write a generic driver, instantiate it *N* times, and have it connected to each of the *N* physical interfaces. You can do this in SystemVerilog by using a virtual interface, which is merely a handle or pointer to a physical interface.[1]

You may need to write a testbench that attaches to several different configurations of your design. In another example, a chip may have multiple configurations. In one, the pins might drive a USB bus, whereas in another the same pins may drive a I2C serial bus. Once again, you can use a virtual interface so that you can decide at run-time which drivers to run in your testbench.

A SystemVerilog interface is more than just signals – you can put executable code inside. This might include routines to read and write to the interface, `initial` and `always` blocks that run code inside the interface, and assertions to constantly check the status of the signals. However, do not put testbench code in an interface. Program blocks have been created expressly for building a testbench, including scheduling their execution in the Reactive region, as described in the SystemVerilog LRM.

---

[1] A better name for a virtual interface would be a "ref interface."

Many of the examples in this chapter can be downloaded from the author's web site: `http://chris.spear.net/systemverilog`

10.1 Virtual Interfaces with the ATM Router

The most common use for a virtual interface is to allow objects in a testbench to refer to items in a replicated interface using a generic handle rather than the actual name. Virtual interfaces are the only mechanism that can bridge the dynamic world of objects with the static world of modules and interfaces.

10.1.1 The Testbench with Just Physical Interfaces

Chap. 4 showed how to build an interface to connect a 4 × 4 ATM router to a testbench. Samples 10.1 and 10.2 show the ATM interfaces for the receive and transmit directions.

Sample 10.1 Rx interface with clocking block

```
// Rx interface with modports and clocking block
interface Rx_if (input logic clk);
  logic [7:0] data;
  logic soc, en, clav, rclk;

  clocking cb @(posedge clk);
    output data, soc, clav;  // Directions are relative
    input  en;               // to the testbench
  endclocking : cb

  modport TB (clocking cb);

  modport DUT (output en, rclk,
               input  data, soc, clav);
endinterface : Rx_if
```

Sample 10.2 Tx interface with clocking block

```
// Tx interface with modports and clocking block
interface Tx_if (input logic clk);
  logic [7:0] data;
  logic soc, en, clav, tclk;

  clocking cb @(posedge clk);
      input  data, soc, en;
      output clav;
  endclocking : cb
```

```
  modport TB (clocking cb);

  modport DUT (output data, soc, en, tclk,
               input  clav);
endinterface : Tx_if
```

These interfaces can be used in a program block shown in Sample 10.3. This procedural code is hard coded with interface names such as `Rx0` and `Tx0`.

Sample 10.3 Testbench using physical interfaces

```
program automatic test(Rx_if.TB Rx0, Rx1, Rx2, Rx3,
                       Tx_if.TB Tx0, Tx1, Tx2, Tx3,
                      input logic clk, output logic rst);

  bit [7:0] bytes[`ATM_SIZE];

  initial begin
    // Reset the device
    rst <= 1;
    Rx0.cb.data <= 0;
    ...
    receive_cell0;
    ...
  end

  task receive_cell0;
    @(Tx0.cb);
    Tx0.cb.clav <= 1;          // Assert ready to receive
    wait (Tx0.cb.soc == 1);    // Wait for Start of Cell

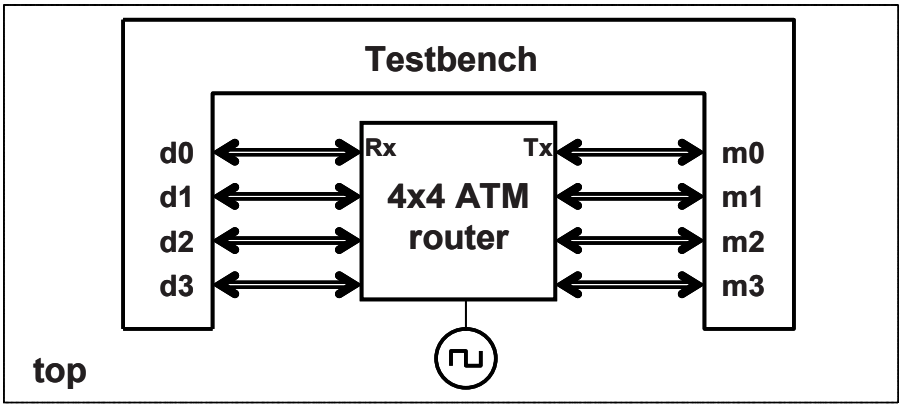
    for (int i=0; i<`ATM_SIZE; i++) begin
      wait (Tx0.cb.en == 0);  // Wait for enable
        @(Tx0.cb);

      bytes[i] = Tx0.cb.data;
      @(Tx0.cb);
      Tx0.cb.clav <= 0;        // Deassert flow control
    end
  endtask

endprogram
```

Figure 10-1 shows the testbench communicating with the design through interfaces.

Figure 10-1 Router and testbench with interfaces



The top level module must connect an array of interfaces to work with the testbench in Sample 10.6. The module in Sample 10.4 instantiates an array of interfaces, and passes this array to the testbench. Since the DUT was written with four RX and four TX interfaces, you need to pass the individual interface array elements into the DUT instance.

Sample 10.4 Top level module with array of interfaces

```
module top;
  logic clk, rst;

  Rx_if Rx[4] (clk);
  Tx_if Tx[4] (clk);

  test       t1 (Rx, Tx, rst);
  atm_router a1 (Rx[0], Rx[1], Rx[2], Rx[3],
                 Tx[0], Tx[1], Tx[2], Tx[3],
                 clk, rst);

  initial begin
    clk = 0;
    forever #20 clk = !clk;
    end
endmodule : top
```

10.1.2 Testbench with Virtual Interfaces

A good OOP technique is to create a class that uses a handle to reference an object, rather than a hard-coded object name. In this case, you can make a single Driver class and a single Monitor class, have them operate on a handle to the data, and then pass in the handle at run-time.

The program block in Sample 10.5 still passed the 4 `Rx` and 4 `Tx` interfaces as ports, as in Sample 10.3, but it creates an array of virtual interfaces, `vRx` and `vTx`. These can now be passed into the constructors for the drivers and monitors.

Sample 10.5 Testbench using virtual interfaces

```
program automatic test(Rx_if.TB Rx0, Rx1, Rx2, Rx3,
                       Tx_if.TB Tx0, Tx1, Tx2, Tx3,
                       output logic rst);

  Driver drv[4];
  Monitor mon[4];
  Scoreboard scb[4];

  virtual Rx_if.TB vRx[4] = Õ{Rx0, Rx1, Rx2, Rx3};
  virtual Tx_if.TB vTx[4] = Õ{Tx0, Tx1, Tx2, Tx3};

  initial begin
    foreach (scb[i]) begin
      scb[i] = new(i);
      drv[i] = new(scb[i].exp_mbx, i, vRx[i]);
      mon[i] = new(scb[i].rcv_mbx, i, vTx[i]);
    end
    ...
  end
endprogram
```

You can also skip the virtual interface array variables, and make an array in the port list. These interfaces can be passed into the constructors as shown in the test in Sample 10.6.

Sample 10.6 Testbench using virtual interfaces

```
program automatic test(Rx_if.TB Rx[4], Tx_if.TB Tx[4],
                       output logic rst);

...
  initial begin
    foreach (scb[i]) begin
      scb[i] = new(i);
      drv[i] = new(scb[i].exp_mbx, i, Rx[i]);
      mon[i] = new(scb[i].rcv_mbx, i, Tx[i]);
    end
    ...
  end
endprogram
```

The driver class in Sample 10.7 looks similar to the code in Sample 10.3, except it uses the virtual interface name `Rx` instead of the physical interface `Rx0`.

Sample 10.7 Driver class using virtual interfaces

```
class Driver;
  int stream_id;
  bit done = 0;
  mailbox exp_mbx;
  virtual Rx_if.TB Rx;

  function new(input mailbox exp_mbx,
               input int     stream_id,
               input virtual Rx_if.TB Rx);
    this.exp_mbx = exp_mbx;
    this.stream_id = stream_id;
    this.Rx = Rx;
  endfunction

  task run(input int ncells, input event driver_done);
    ATM_Cell ac;

    fork // Spawn this as a separate thread
      begin
        // Initialize output signals
        Rx.cb.clav <= 0;
        Rx.cb.soc <= 0;
        @Rx.cb;

        // Drive cells until the last one is sent
        repeat (ncells) begin
          ac = new
          assert(ac.randomize);
          if (ac.eot_cell) break;   // End transmission
            drive_cell(ac);
       end

        $display("@%0t: Driver::run Driver[%0d] is done",
                 $time, stream_id);
        -> driver_done;
      end
    join_none
  endtask : run

  task drive_cell(input ATM_Cell ac);
    bit [7:0] bytes[];

    #ac.delay;
    ac.byte_pack(bytes);

    $display("@%0t: Driver::drive_cell(%0d) vci=%h",
             $time, stream_id, ac.vci);
```

```
    // Wait to start on a new cycle
    @Rx.cb;
    Rx.cb.clav <= 1;            // Assert ready to xfr
    do
      @Rx.cb;
    while (Rx.cb.en != 0)       // Wait for enable low

    Rx.cb.soc <= 1;             // Start of cell
    Rx.cb.data <= bytes[0];     // Drive first byte
    @Rx.cb;
    Rx.cb.soc <= 0;             // Start of cell done
    Rx.cb.data <= bytes[1];     // Drive first byte

    for (int i=2; i<`ATM_SIZE; i++) begin
      @Rx.cb;
      Rx.cb.data <= bytes[i];
    end

    @Rx.cb;
    Rx.cb.soc <= 1'bz;          // Tristate SOC at end
    Rx.cb.clav <= 0;
    Rx.cb.data <= 8'bz;         // Clear data lines
    $display("@%0t: Driver::drive_cell(%0d) finish",
             $time, stream_id);

    // Send cell to scoreboard
    exp_mbx.put(ac);

  endtask : drive_cell_t

endclass : Driver
```

10.1.3 Connecting the Testbench to an Interface in Port List

This book shows tests that connect to the DUT with interfaces in the port list. This style is comfortable to Verilog users who have always connected modules using signals in ports. Sample 10.8 is the top level module, also known as a test harness, which connects the DUT and test using an interface in the port list.

Sample 10.8 Test harness using an interface in the port list

```
module top;
  bus_ifc bus();   // Instantiate the interface
  test t1(bus);    // Pass to test through port list
  dut  d1(bus);    // Pass to DUT through port list
  ...
endmodule
```

Sample 10.9 shows the program block with an interface in the port list.

Sample 10.9 Test with an interface in the port list

```
program automatic test(bus_ifc bus);
  initial $display(bus.data);  // Use an interface signal
endprogram
```

What happens if you add a new interface to your design? The test harness in Sample 10.10 declares the new bus and put it in the port lists.

Sample 10.10 Top module with a second interface in the test's port list

```
module top;
  bus_ifc bus();      // Instantiate the interface
  new_ifc newb();     // and a new one
  test t1(bus, newb); // Test with two interfaces
  dut  d1(bus, newb); // DUT with two interfaces
  ...
endmodule
```

Now you have to change the test in Sample 10.9 to include another interface in the port list, giving the test in Sample 10.11.

Sample 10.11 Test with two interfaces in the port list

```
program automatic test(bus_ifc bus, new_ifc newb);
  initial $display(bus.data);  // Use an interface signal
endprogram
```

Adding a new interface to your design means you need to edit all existing tests so that they can plug into the test harness. How can you avoid this extra work? Avoid port connections!

10.1.4 Connecting the Test to an Interface with an XMR

Your test needs to connect to the physical interface in the harness, and so use a cross module reference (XMR) and a virtual interface in the program block as shown in Sample 10.12. You must use a virtual interface so that you can assign it the physical interface in the top level module.

Sample 10.12 Test with virtual interface and XMR

```
program automatic test();
  virtual bus_ifc bus = top.bus; // Cross module reference
  initial $display(bus.data);    // Use an interface signal
endprogram
```

The program connects to the test harness shown in Sample 10.13.

Sample 10.13 Test harness without interfaces in the port list

```
module top;
  bus_ifc bus();   // Instantiate the interface
  test t1();       // DonÕt use port list for test
  dut  d1(bus);    // Still use port list for DUT
  ...
endmodule
```

This approach is recommended by methodologies such as the VMM to make your test code more reusable. If you add a new interface to your design, as shown in Sample 10.14, the test harness changes, but existing tests don't have to change.

Sample 10.14 Test harness with a second interface

```
module top;
  bus_ifc bus();      // Instantiate the interface
  new_ifc newb();     // and a new one
  test t1();          // Instantiaton remains the same
  dut  d1(bus, newb);
  ...
endmodule
```

The harness in Sample 10.14 works with the test in Sample 10.12 that does not know about the new interface, as well as the test in Sample 10.15 that does.

Sample 10.15 Test with two virtual interfaces and XMRs

```
program automatic test();
  virtual bus_ifc bus = top.bus;
  virtual new_ifc newb = top.newb

  initial begin
    $display(bus.data);  // Use existing interface
    $display(newb.addr); // and new one
  end
endprogram
```



Some methodologies have a rule that makes the connection between tests and harnesses slightly more complicated than with traditional ports, but means you don't have to modify existing tests, even if the design changes. The examples in this book use the simple style of interfaces in the port lists, but you should decide if test reuse is important enough to change your coding style.

10.2 Connecting to Multiple Design Configurations

A common challenge to verifying a design is that it may have several configurations. You could make a separate testbench for each configuration, but this could lead to a combinatorial explosion as you explore every alternative. Instead, you can use virtual interfaces to dynamically connect to the optional interfaces.

10.2.1 A Mesh Design

Sample 10.16 is built of a simple replicated component, an 8-bit counter. This resembles a DUT that has a device such as a network chip or processor that is instantiated repeatedly in a mesh configuration. The key idea is that the top-level netlist creates an array of interfaces and counters. Now the testbench can connect its array of virtual interfaces to the physical ones.

Sample 10.16 shows the code for the counter's interface, `X_if`. It has a `$strobe` in an `always` block to print the signal values.

Sample 10.16 Interface for 8-bit counter

```
interface X_if (input logic clk);
  logic [7:0] din, dout;
  logic reset_l, load;

  clocking cb @(posedge clk);
    output din, load;
    input dout;
  endclocking

  always @cb
    $strobe("@%0t: %m: out=%0d, in=%0d, ld=%0d, r=%0d",
            $time, dout, din, load, reset_l);

  modport DUT (input clk, din, reset_l, load,
               output dout);

  modport TB (clocking cb, output reset_l);
endinterface
```

The simple counter is shown in Sample 10.17.

Sample 10.17 Counter model using `X_if` interface

```
// Simple 8-bit counter with load and active-low reset
module dut(X_if.DUT xi);
  logic [7:0] count;
  assign xi.dout = count;

  always @(posedge xi.clk or negedge xi.reset_l)
    begin
      if (!xi.reset_l)  count <= 0;
      else if (xi.load) count <= xi.din;
      else              count <= count+1;
    end
endmodule
```

The top-level netlist in Sample 10.18 uses a generate statement to instantiate `NUM_XI` interfaces and counters, but only one testbench.

Sample 10.18 Testbench using an array of virtual interfaces

```
parameter NUM_XI = 2;  // Number of design instances

module top;
  // Clock generator
  bit clk;
  initial begin
    clk <= '0;
    forever #20 clk = ~clk;
  end

  // Instantiate N interfaces
  X_if xi[NUM_XI] (clk);

  // Instantiate the testbench
  test tb();

  // Generate N DUT instances
  generate
  for (genvar i=0; i<NUM_XI; i++)
    begin : dut_blk
      dut d (xi[i]);
    end
  endgenerate

endmodule : top
```

In Sample 10.19, the key line in the testbench is where the local virtual interface array, `vxi`, is assigned to point to the array of physical interfaces in the top module, `top.xi`. (Note that this example takes some shortcuts compared to the recommendations in Chap. 8. To simplify Sample 10.18, the environment class has been merged with the test, whereas the generator, agent, and driver layers have been compressed into the driver.)

The testbench assumes there is at least one counter and thus at least one X interface. If your design could have zero counters, you would have to use a dynamic array to hold the virtual interfaces, as a fixed-size array cannot have a size of zero.

Sample 10.19 Counter testbench using virtual interfaces

```
program automatic test;

  virtual X_if.TB vxi[NUM_XI]; // Virtual interface array
  Driver driver[];

  initial begin
    // Connect local virtual interface to top
    vxi = top.xi;

    // Create N drivers
    driver = new[NUM_XI];
    foreach (driver[i])
      driver[i] = new(vxi[i], i);

    foreach (driver[i]) begin
      int j = i
      fork
        begin
          driver[j].reset();
          driver[j].load_op();
        end
      join_none

    repeat (10) @(vxi[0].cb);
  end

endprogram
```

Of course in this simple example, you could just pass the interface directly into the `Driver`'s constructor, rather than make a separate variable.

In Sample 10.20, the `Driver` class uses a single virtual interface to drive and sample signals from the counter.

Sample 10.20 `Driver` class using virtual interfaces

```
class Driver;
  virtual X_if xi;
  int id;

  function new(input virtual X_if.TB xi, input int id);
    this.xi = xi;
    this.id = id;
  endfunction

  task reset();
    $display("@%0t: %m: Start reset [%0d]",
             $time, id);
    // Reset the device
    xi.reset_l <= 1;
    xi.cb.load <= 0;
    xi.cb.din <= 0;
    @(xi.cb) xi.reset_l <= 0;
    @(xi.cb) xi.reset_l <= 1;
    $display("@%0t: %m: End reset [%0d]",
             $time, id);
  endtask : reset

  task load_op();
    $display("@%0t: %m: Start load [%0d]",
             $time, id);
    ##1 xi.cb.load <= 1;
    xi.cb.din <= id + 10;

    ##1 xi.cb.load <= 0;
    repeat (5) @(xi.cb);
    $display("@%0t: %m: End load [%0d]",
             $time, id);
  endtask : load_op

endclass : Driver
```

10.2.2 Using `typedefs` with Virtual Interfaces

You can reduce the amount of typing, and ensure you always use the correct modport by replacing "`virtual X_if.TB`" with a `typedef`, as shown in Samples 10.21 and 10.22 of the testbench and driver.

Sample 10.21 Testbench using a `typedef` for virtual interfaces

```
typedef virtual X_if.TB vx_if;

program automatic test;
  vx_if vxi[NUM_XI];      // Virtual interface array
  Driver driver[];
  ...
endprogram
```

Sample 10.22 Driver using a `typedef` for virtual interfaces

```
class Driver;
  vx_if xi;
  int id;

  function new(input vx_if xi, input int id);
    this.xi = xi;
    this.id = id;
  endfunction
  ...
endclass : Driver
```

10.2.3 Passing Virtual Interface Array Using a Port

The previous examples passed the array of virtual interfaces using a cross module reference (XMR). An alternative is to pass the array in a port. Since the array in the top netlist is static and so only needs to be referenced once, the XMR style makes more sense than using a port that normally is used to pass changing values.

Sample 10.23 uses a global parameter to define the number of X interfaces. Here is a snippet of the top netlist.

Sample 10.23 Testbench using an array of virtual interfaces

```
parameter NUM_XI = 2;  // Number of instances

module top;
  // Instantiate N interfaces
  X_if xi [NUM_XI] (clk);

  ...
  // Instantiate the testbench
  test tb(xi);

endmodule : top
```

The testbench that uses the virtual interfaces is shown in Sample 10.24. It creates an array of virtual interfaces so that it can pass them into the constructor for the driver class, or just pass the interface directly from the port.

Sample 10.24 Testbench passing virtual interfaces with a port

```
program automatic test(X_if xi [NUM_XI]);

  Driver driver[];
  virtual X_if vxi[NUM_XI];

  initial begin
    // Connect the local virtual interfaces to the top
    if (NUM_XI <= 0) $finish;

    driver = new[NUM_XI];
    vxi = xi;    // Assign the interface array

    for (int i=0; i<NUM_XI; i++) begin
      driver[i] = new(vxi[i], i);
      driver[i].reset;
    end
  ...
  end

endprogram
```

10.3 Procedural Code in an Interface

Just as a class contains both variables and routines, an interface can contain code such as routines, assertions, and `initial` and `always` blocks. Recall that an interface includes the signals and functionality of the communication between two blocks. So the interface block for a bus can contain the signals and also routines to perform commands such as a read or write. The inner workings of these routines are hidden from the external blocks, allowing you to defer the actual implementation. Access to these routines is controlled using the `modport` statement, just as with signals. A task or function is imported into a `modport` so that it is then visible to any block that uses the `modport`.

These routines can be used by both the design and the testbench. This approach ensures that both are using the same protocol, eliminating a common source of testbench bugs. However, not all synthesis tools can handle routines in an assertion.

You can verify a protocol with assertions in an interface. An assertion can check for illegal combinations, such as protocol violations and unknown values. These can display state information and stop simulation immediately so that you can easily debug the problem. An assertion can also fire when good transactions occur.

Functional coverage code will use this type of assertion to trigger the gathering of coverage information.

10.3.1 Interface with Parallel Protocol

When creating your system, you may not know whether to choose a parallel or serial protocol. The interface in Sample 10.25 has two tasks, `initiatorSend` and `targetRcv`, that send a transaction between two blocks using the interface signals. It sends the address and data in parallel across two 8-bit buses.

Sample 10.25 Interface with tasks for parallel protocol

```
interface simple_if(input logic clk);
  logic [7:0] addr;
  logic [7:0] data;
  bus_cmd_e cmd;
  modport TARGET
    (input  addr, cmd, data,
     import task targetRcv (output bus_cmd_e c,
                            logic [7:0] a, d));
   modport INITIATOR
     (output addr, cmd, data,
      import task initiatorSend(input bus_cmd_e c,
                                logic [7:0] a, d)
      );

  // Parallel send
  task initiatorSend(input bus_cmd_e c,
                           logic [7:0] a, d);
    @(posedge clk);
    cmd <= c;
    addr <= a;
    data <= d;
  endtask

  // Parallel receive
  task targetRcv(output bus_cmd_e c, logic [7:0] a, d);
    @(posedge clk);
    a = addr;
    d = data;
    c = cmd;
  endtask
endinterface: simple_if
```

10.3.2 Interface with Serial Protocol

The interface in Sample 10.26 implements a serial interface for sending and receiving the address and data values. It has the same interface and routine names as Sample 10.25, and so you can swap between the two without having to change any design or testbench code.

Sample 10.26 Interface with tasks for serial protocol

```
interface simple_if(input logic clk);
  logic addr;
  logic data;
  logic start = 0;
  bus_cmd_e cmd;

  modport TARGET(input  addr, cmd, data,
                 import task targetRcv (output bus_cmd_e c,
                                        logic [7:0] a, d));
  modport INITIATOR(output addr, cmd, data,
                    import task initiatorSend(input bus_cmd_e c,
                                              logic [7:0] a, d));

  // Serial send
  task initiatorSend(input bus_cmd_e c, logic [7:0] a, d);
    @(posedge clk);
    start <= 1;
    cmd <= c;
    foreach (a[i]) begin
      addr <= a[i];
      data <= d[i];
      @(posedge clk);
      start <= 0;
    end
    cmd <= IDLE;
  endtask

  // Serial receive
  task targetRcv(output bus_cmd_e c, logic [7:0] a, d);
    @(posedge start);
    c = cmd;
    foreach (a[i]) begin
      @(posedge clk);
      a[i] = addr;
      d[i] = data;
    end
  endtask

endinterface: simple_if
```

10.3.3 Limitations of Interface Code

Tasks in interfaces are fine for RTL, where the functionality is strictly defined. However, these tasks are a poor choice for any type of verification IP. Interfaces and their code cannot be extended, overloaded, or dynamically instantiated based on configuration. An interface cannot have private data. Any code for verification needs maximum flexibility and configurability, and so should go in classes that run in a program block.

10.4 Conclusion

The interface construct in SystemVerilog provides a powerful technique to group together the connectivity, timing, and functionality for the communication between blocks. In this chapter you saw how you can create a single testbench that connects to many different design configurations containing multiple interfaces. Your signal layer code can connect to a variable number of physical interfaces at run-time with virtual interfaces. Additionally, an interface can have routines that drives the signals and assertions to check the protocol, but put the test in a program block, not an interface.

In many ways, an interface can resemble a class with pointers, encapsulation, and abstraction. This lets you create an interface to model your system at a higher level than Verilog's traditional ports and wires. Just remember to keep the testbench in the program block.

Chapter 11

A Complete SystemVerilog Testbench

This chapter applies the many concepts you have learned about SystemVerilog features to verify a design. The testbench creates constrained random stimulus, and gathers functional coverage. It is structured according to the guidelines from Chap. 8 and so you can inject new behavior without modifying the lower-level blocks.

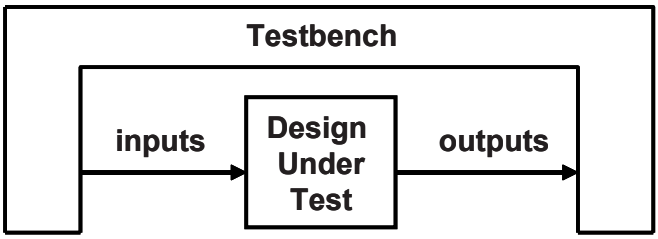
The design is an ATM switch that was shown in Sutherland et al. (2006), who based his SystemVerilog description on an example from Janick Bergeron's Verification Guild. Sutherland took the original Verilog design and used SystemVerilog design features to create a switch that can be configured from 4× 4 to 16× 16. The testbench in the original example creates ATM cells using `$urandom`, overwrites certain fields with ID values, sends them through the device, and then checks that the same values were received.

The entire example, with the testbench and ATM switch, is available for download at `http://chris.spear.net/systemverilog`. This chapter shows just the testbench code.

11.1 Design Blocks

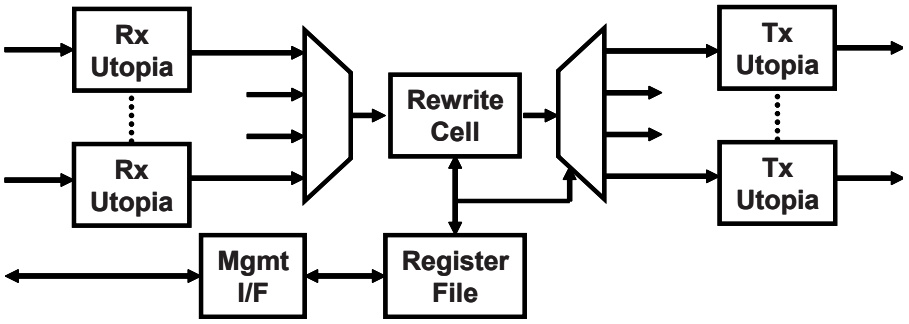
The overall connection between the design and testbench, shown in Figure 11-1, follows the pattern shown in Chap. 4.

Figure 11-1 The testbench – design environment



The top level of the design is called `squat`, as shown in Figure 11-2. The module has 1..*N* Utopia Rx interfaces that are sending UNI-formatted cells. Inside the DUT, cells are stored, converted to NNI format, and forwarded to the Tx interfaces. The forwarding is done according to a lookup table that is addressed with the VPI field of the incoming cell. The table is programmed through the management interface.

Figure 11-2 Block diagram for the squat design



The top level module in Sample 11.1 defines arrays of interfaces for the Rx and Tx ports.

Sample 11.1 Top level module

```
`timescale 1ns/1ns
`define TxPorts 4  // set number of transmit ports
`define RxPorts 4  // set number of receive ports

module top;
  parameter int NumRx = `RxPorts;
  parameter int NumTx = `TxPorts;

  logic rst, clk;
  // System Clock and Reset
  initial begin
    rst = 0; clk = 0;
    #5ns rst = 1;
    #5ns clk = 1;
    #5ns rst = 0; clk = 0;
    forever
      #5ns clk = ~clk;
  end

  Utopia Rx[0:NumRx-1] ();// NumRx x Level 1 Utopia Rx Interface
  Utopia Tx[0:NumTx-1] ();// NumTx x Level 1 Utopia Tx Interface
  cpu_ifc mif();          // Utopia management interface
  squat #(NumRx, NumTx) squat(Rx, Tx, mif, rst, clk);   // DUT
  test  #(NumRx, NumTx) t1(Rx, Tx, mif, rst, clk);      // Test
endmodule : top
```

The testbench program in Sample 11.2 passes the interfaces and signals through the port list. See Section 10.1.4 for a discussion on posts vs. cross module references. The actual testbench code is in the `Environment` class. The program steps through the phases of the environment.

Sample 11.2 Testbench program

```
program automatic test
  #(parameter int NumRx = 4, parameter int NumTx = 4)
   (Utopia.TB_Rx Rx[0:NumRx-1],
    Utopia.TB_Tx Tx[0:NumTx-1],
    cpu_ifc.Test mif,
    input logic rst, clk);

`include "environment.sv"
  Environment env;

  initial begin
    env = new(Rx, Tx, NumRx, NumTx, mif);
    env.gen_cfg();
    env.build();
    env.run();
    env.wrap_up();
  end
endprogram // test
```

The testbench loads control information into the ATM switch through the Management interface, also known as the CPU interface, shown in Sample 11.3. In this chapter's examples, the interface is only used to load the lookup table that maps VPI to forwarding masks.

Sample 11.3 CPU Management Interface

```
interface cpu_ifc;
  logic        BusMode, Sel, Rd_DS, Wr_RW, Rdy_Dtack;
  logic [11:0] Addr;
  CellCfgType  DataIn, DataOut;    // Defined in Sample 11.11

  modport Peripheral
          (input  BusMode, Addr, Sel, DataIn, Rd_DS, Wr_RW,
           output DataOut, Rdy_Dtack);

  modport Test
          (output BusMode, Addr, Sel, DataIn, Rd_DS, Wr_RW,
           input  DataOut, Rdy_Dtack);

endinterface : cpu_ifc

typedef virtual cpu_ifc.Test vCPU_T;
```

Sample 11.4 shows the Utopia interface, which is used by the testbench to communicate with the squat design by transmitting and receiving ATM cells. The interface has clocking blocks for the transmit and receive paths, and modports for the design and testbench connections to the interface.

Sample 11.4 Utopia interface

```
interface Utopia;
  parameter int IfWidth = 8;

  logic [IfWidth-1:0] data;
  bit clk_in, clk_out;
  bit soc, en, clav, valid, ready, reset, selected;

  ATMCellType ATMcell;  // union of structures for ATM cells

  modport TopReceive (
    input  data, soc, clav,
    output clk_in, reset, ready, clk_out, en, ATMcell, valid );

  modport TopTransmit (
    input  clav,
    inout  selected,
    output clk_in, clk_out, ATMcell, data, soc, en, valid,
reset, ready );

  modport CoreReceive (
    input  clk_in, data, soc, clav, ready, reset,
    output clk_out, en, ATMcell, valid );

  modport CoreTransmit (
    input  clk_in, clav, ATMcell, valid, reset,
    output clk_out, data, soc, en, ready );

   clocking cbr @(negedge clk_out);
      input clk_in, clk_out, ATMcell, valid, reset, en, ready;
      output data, soc, clav;
   endclocking : cbr
   modport TB_Rx (clocking cbr);

   clocking cbt @(negedge clk_out);
      input  clk_out, clk_in, ATMcell, soc, en, valid,
             reset, data, ready;
      output clav;
   endclocking : cbt
   modport TB_Tx (clocking cbt);

endinterface

typedef virtual Utopia vUtopia;
typedef virtual Utopia.TB_Rx vUtopiaRx;
typedef virtual Utopia.TB_Tx vUtopiaTx;
```

11.2 Testbench Blocks

The environment class, as shown in Section 8.2.1, is the scaffolding that supports the testbench structure. Inside this class lies the blocks of your layered testbench, such as generators, drivers, monitors, and scoreboard. The environment also controls the sequencing of the four testbench steps: generate a random configuration, build the testbench environment, run the test and wait for it to complete, and a wrap-up phase to shut down the system and generate reports.

Sample 11.5 Environment class header

```
class Environment;
  UNI_generator gen[];
  mailbox gen2drv[];
  event   drv2gen[];
  Driver drv[];
  Monitor mon[];
  Config cfg;
  Scoreboard scb;
  Coverage cov;
  virtual Utopia.TB_Rx Rx[];
  virtual Utopia.TB_Tx Tx[];
  int numRx, numTx;
  vCPU_T mif;
  CPU_driver cpu;

  extern function new(input vUtopiaRx Rx[],
                      input vUtopiaTx Tx[],
                      input int numRx, numTx,
                      input vCPU_T mif);
  extern virtual function void gen_cfg();
  extern virtual function void build();
  extern virtual task run();
  extern virtual function void wrap_up();

endclass : Environment
```

With the `$test$plusargs()` system task, the `Environment` class constructor in Sample 11.6 looks for the VCS switch `+ntb_random_seed`, which sets the random seed for the simulation. The system task `$value$plusargs()` extracts the value from the switch. Your simulator may have a different way to set the seed. It is important to print the seed in the log file so that if the test fails, you can run it again with the same value.

Sample 11.6 Environment class methods

```
//---------------------------------------------------------------
// Construct an environment instance
function Environment::new(input vUtopiaRx Rx[],
                          input vUtopiaTx Tx[],
                          input int numRx, numTx,
                          input vCPU_T mif);
  this.Rx = new[Rx.size()];
  foreach (Rx[i]) this.Rx[i] = Rx[i];
  this.Tx = new[Tx.size()];
  foreach (Tx[i]) this.Tx[i] = Tx[i];
  this.numRx = numRx;
  this.numTx = numTx;
  this.mif = mif;
  cfg = new(NumRx,NumTx);

  if ($test$plusargs("ntb_random_seed")) begin
    int seed;
    $value$plusargs("ntb_random_seed=%d", seed);
    $display("Simulation run with random seed=%0d", seed);
  end
  else
    $display("Simulation run with default random seed");
endfunction : new

//---------------------------------------------------------------
// Randomize the configuration descriptor
function void Environment::gen_cfg();
  assert(cfg.randomize());
  cfg.display();
endfunction : gen_cfg

//---------------------------------------------------------------
// Build the environment objects for this test
// Note that objects are built for every channel,
// even if they are not used.  This reduces null handle bugs.
function void Environment::build();
  cpu = new(mif, cfg);
  gen = new[numRx];
  drv = new[numRx];
  gen2drv = new[numRx];
  drv2gen = new[numRx];
  scb = new(cfg);
  cov = new();

  // Build generators
  foreach(gen[i]) begin
    gen2drv[i] = new();
```

```
    gen[i] = new(gen2drv[i], drv2gen[i],
                 cfg.cells_per_chan[i], i);
    drv[i] = new(gen2drv[i], drv2gen[i], Rx[i], i);
  end

  // Build monitors
  mon = new[numTx];
  foreach (mon[i])
    mon[i] = new(Tx[i], i);

  // Connect scoreboard to drivers & monitors with callbacks
  begin
    Scb_Driver_cbs sdc = new(scb);
    Scb_Monitor_cbs smc = new(scb);
    foreach (drv[i]) drv[i].cbsq.push_back(sdc);
    foreach (mon[i]) mon[i].cbsq.push_back(smc);
  end

  // Connect coverage to monitor with callbacks
  begin
    Cov_Monitor_cbs smc = new(cov);
   foreach (mon[i])
     mon[i].cbsq.push_back(smc);
  end
endfunction : build

//-------------------------------------------------------------
// Start the transactors: generators, drivers, monitors
// Channels that are not in use don't get started
task Environment::run();
  int num_gen_running;

  // The CPU interface initializes before anyone else
  cpu.run();

  num_gen_running = numRx;

  // For each input RX channel, start generator and driver
  foreach(gen[i]) begin
    int j=i;     // Automatic var holds index in spawned threads
    fork
      begin
        if (cfg.in_use_Rx[j])
          gen[j].run();          // Wait for generator to finish
        num_gen_running--;// Decrement driver count
     end
     if (cfg.in_use_Rx[j]) drv[j].run();
    join_none
  end
```

```
  // For each output TX channel, start monitor
  foreach(mon[i]) begin
    int j=i;     // Automatic var holds index in spawned threads
    fork
      mon[j].run();
    join_none
  end

  // Wait for all generators to finish, or time-out
  fork : timeout_block
    wait (num_gen_running == 0);
    begin
      repeat (1_000_000) @(Rx[0].cbr);
      $display("@%0t: %m ERROR: Generator timeout ", $time);
      cfg.nErrors++;
    end
  join_any
  disable timeout_block;

  // Wait for the data to flow through switch, into monitors,
  // and scoreboards
  repeat (1_000) @(Rx[0].cbr);
endtask : run

//---------------------------------------------------------------
// Post-run cleanup / reporting
function void Environment::wrap_up();
  $display("@%0t: End of sim, %0d errors, %0d warnings",
           $time, cfg.nErrors, cfg.nWarnings);
  scb.wrap_up;
endfunction : wrap_up
```

The method `Environment::build` in Sample 11.6 connects the scoreboard to the driver and monitor with the callback class, which is shown in Sample 11.7, `Scb_Driver_cbs`. This class sends the expected values to the scoreboard. The base driver callback class, `Driver_cbs`, is shown in Sample 11.20.

Sample 11.7 Callback class connects driver and scoreboard

```
class Scb_Driver_cbs extends Driver_cbs;
  Scoreboard scb;

  function new(input Scoreboard scb);
    this.scb = scb;
  endfunction : new

  // Send received cell to scoreboard
  virtual task post_tx(input Driver drv,
    input UNI_cell cell);
    scb.save_expected(cell);
  endtask : post_tx
endclass : Scb_Driver_cbs
```

The callback class in Sample 11.8, `Scb_Monitor_cbs`, connects the monitor with the scoreboard. The base monitor callback class, `Monitor_cbs`, is shown in Sample 11.21.

Sample 11.8 Callback class connects monitor and scoreboard

```
class Scb_Monitor_cbs extends Monitor_cbs;
  Scoreboard scb;

  function new(input Scoreboard scb);
    this.scb = scb;
  endfunction : new

  // Send received cell to scoreboard
  virtual task post_rx(input Monitor mon,
                       input NNI_cell cell);
    scb.check_actual(cell, mon.PortID);
  endtask : post_rx
endclass : Scb_Monitor_cbs
```

The environment connects the monitor to the coverage class with the final callback class, `Cov_Monitor_cbs`, shown in Sample 11.9.

Sample 11.9 Callback class connects the monitor and coverage

```
class Cov_Monitor_cbs extends Monitor_cbs;
  Coverage cov;

  function new(input Coverage cov);
    this.cov = cov;
  endfunction : new

  // Send received cell to coverage
  virtual task post_rx(input Monitor mon,
                       input NNI_cell cell);
    CellCfgType CellCfg = top.squat.lut.read(cell.VPI);
    cov.sample(mon.PortID, CellCfg.FWD);
  endtask : post_rx
endclass : Cov_Monitor_cbs
```

The random configuration class header is shown in Sample 11.10. It starts with `nCells`, a random value for the total number of cells that flow through the system. The constraint `c_nCells_valid` ensures the number of cells is valid by being greater than zero, whereas `c_nCells_reasonable` limits the test to a reasonable size, 1,000 cells. You can disable or override this if you want longer tests.

Next is a dynamic bit array, `in_use_Rx`, to specify which Rx channels into the switch are active. This is used in Sample 11.6 in the `run` method so that only active channels run.

The array `cells_per_chan` is used to randomly divide the total number of cells across the active channels. The constraint `zero_unused_channels` sets the number of cells to zero for inactive channels. To help the solver, the active channel mask is solved before dividing up the cells between channels. Otherwise, a channel would be inactive only if the number of cells assigned to it was zero, which is very unlikely.

Sample 11.10 Environment configuration class

```
class Config;
  int nErrors, nWarnings;  // Number of errors, warnings
  bit [31:0] numRx, numTx; // Copy of parameters

  rand bit [31:0] nCells;  // Total cells
  constraint c_nCells_valid
     {nCells > 0; }
  constraint c_nCells_reasonable
     {nCells < 1000; }

  rand bit in_use_Rx[];     // Input / output channel enabled
  constraint c_in_use_valid
    {in_use_Rx.sum > 0; }   // At least one RX is enabled

  rand bit [31:0] cells_per_chan[];
  constraint c_sum_ncells_sum  // Split cells over all channels
    {cells_per_chan.sum == nCells;}    // Total number of cells

  // Set the cell count to zero for any channel not in use
  constraint zero_unused_channels
    {foreach (cells_per_chan[i])
      {
       // Needed for even dist of in_use
       solve in_use_Rx[i] before cells_per_chan[i];
       if (in_use_Rx[i])
         cells_per_chan[i] inside {[1:nCells]};
         else cells_per_chan[i] == 0;
       }
    }

  extern function new(input bit [31:0] numRx, numTx);
  extern virtual function void display(input string prefix="");
endclass : Config
```

The cell rewriting and forwarding configuration type is shown in Sample 11.11.

Sample 11.11 Cell configuration type

```
typedef struct packed {
  bit [`TxPorts-1:0] FWD;
  bit [11:0] VPI;
} CellCfgType;
```

The methods for the configuration class are shown in Sample 11.12

Sample 11.12 Configuration class methods

```
function Config::new(input bit [31:0] numRx, numTx);
  this.numRx = numRx;
  in_use_Rx = new[numRx];
  this.numTx = numTx;
  cells_per_chan = new[numRx];
endfunction : new

function void Config::display(input string prefix);
  $write("%sConfig: numRx=%0d, numRx=%0d, nCells=%0d (",
         prefix, numRx, numRx, nCells);
  foreach (cells_per_chan[i])
    $write("%0d ", cells_per_chan[i]);
  $write("), enabled RX: ", prefix);
  foreach (in_use_Rx[i]) if (in_use_Rx[i]) $write("%0d ", i);
  $display;
 endfunction : display
```

The ATM switch accepts UNI-formatted cells and sends out NNI formatted cells. These cells are sent through both an OOP testbench and a structural design, and so they are defined using `typedef`. The major difference between the two formats is that the UNI's GFC and VPI field are merged into the NNI's VPI. The definitions in Sample 11.13 through 11.11 from Sutherland et al. (2006).

Sample 11.13 UNI cell format

```
typedef struct packed {
  bit        [3:0]  GFC;
  bit        [7:0]  VPI;
  bit        [15:0] VCI;
  bit               CLP;
  bit        [2:0]  PT;
  bit        [7:0]  HEC;
  bit [0:47] [7:0]  Payload;
} uniType;
```

Sample 11.14 NNI cell format

```
typedef struct packed {
  bit        [11:0] VPI;
  bit        [15:0] VCI;
  bit               CLP;
  bit        [2:0]  PT;
  bit        [7:0]  HEC;
  bit [0:47] [7:0]  Payload;
} nniType;
```

The UNI and NNI cells are merged with a byte memory to form a universal type, shown in Sample 11.15.

Sample 11.15 ATMCellType

```
typedef union packed {
  uniType uni;
  nniType nni;
  bit [0:52] [7:0] Mem;
} ATMCellType;
```

The testbench generates constrained random ATM cells, shown in Sample 11.16, that are extended from the `BaseTr` class, defined in Sample 8.26

Sample 11.16 UNI_cell definition

```
class UNI_cell extends BaseTr;
  // Physical fields
  rand bit        [3:0]  GFC;
  rand bit        [7:0]  VPI;
  rand bit        [15:0] VCI;
  rand bit               CLP;
  rand bit        [2:0]  PT;
       bit        [7:0]  HEC;
  rand bit [0:47] [7:0]  Payload;

  // Meta-data fields
  static bit [7:0] syndrome[0:255];
  static bit syndrome_not_generated = 1;

  extern function new();
  extern function void post_randomize();
  extern virtual function bit compare(input BaseTr to);
  extern virtual function void display(input string prefix="");
  extern virtual function void copy_data(input UNI_cell copy);
  extern virtual function BaseTr copy(input BaseTr to=null);
  extern virtual function void pack(output ATMCellType to);
  extern virtual function void unpack(input ATMCellType from);
  extern function NNI_cell to_NNI();
  extern function void generate_syndrome();
  extern function bit [7:0] hec (bit [31:0] hdr);
endclass : UNI_cell
```

Sample 11.17 shows the methods for the UNI cell.

Sample 11.17 UNI_cell methods

```
function UNI_cell::new();
  if (syndrome_not_generated)
    generate_syndrome();
endfunction : new


// Compute the HEC value after all other data has been chosen
function void UNI_cell::post_randomize();
  HEC = hec({GFC, VPI, VCI, CLP, PT});
endfunction : post_randomize


// Compare this cell with another
// This could be improved by telling what field mismatched
function bit UNI_cell::compare(input BaseTr to);
  UNI_cell cell;
  $cast(cell, to);
  if (this.GFC != cell.GFC)         return 0;
  if (this.VPI != cell.VPI)         return 0;
  if (this.VCI != cell.VCI)         return 0;
  if (this.CLP != cell.CLP)         return 0;
  if (this.PT  != cell.PT)          return 0;
  if (this.HEC != cell.HEC)         return 0;
  if (this.Payload != cell.Payload) return 0;
  return 1;
endfunction : compare


// Print a Òprettyó version of this object
function void UNI_cell::display(input string prefix);
  ATMCellType p;

  $display("%sUNI id:%0d GFC=%x, VPI=%x, VCI=%x, CLP=%b, PT=%x,
HEC=%x, Payload[0]=%x",
           prefix, id, GFC, VPI, VCI, CLP, PT, HEC, Payload[0]);
  this.pack(p);
  $write("%s", prefix);
  foreach (p.Mem[i]) $write("%x ", p.Mem[i]);
  $display;
endfunction : display


// Copy the data fields of this cell
function void UNI_cell::copy_data(input UNI_cell copy);
  copy.GFC     = this.GFC;
  copy.VPI     = this.VPI;
  copy.VCI     = this.VCI;
```

```
  copy.CLP     = this.CLP;
  copy.PT      = this.PT;
  copy.HEC     = this.HEC;
  copy.Payload = this.Payload;
endfunction : copy_data


// Make a copy of this object
function BaseTr UNI_cell::copy(input BaseTr to);
  UNI_cell dst;
  if (to == null) dst = new();
  else            $cast(dst, to);
  copy_data(dst);
  return dst;
endfunction : copy


// Pack this objectÕs properties into a byte array
function void UNI_cell::pack(output ATMCellType to);
  to.uni.GFC     = this.GFC;
  to.uni.VPI     = this.VPI;
  to.uni.VCI     = this.VCI;
  to.uni.CLP     = this.CLP;
  to.uni.PT      = this.PT;
  to.uni.HEC     = this.HEC;
  to.uni.Payload = this.Payload;
endfunction : pack


// Unpack a byte array into this object
function void UNI_cell::unpack(input ATMCellType from);
  this.GFC     = from.uni.GFC;
  this.VPI     = from.uni.VPI;
  this.VCI     = from.uni.VCI;
  this.CLP     = from.uni.CLP;
  this.PT      = from.uni.PT;
  this.HEC     = from.uni.HEC;
  this.Payload = from.uni.Payload;
endfunction : unpack


// Generate a NNI cell from an UNI cell - used in scoreboard
function NNI_cell UNI_cell::to_NNI();
  NNI_cell copy;
  copy = new();
  copy.VPI     = this.VPI;   // NNI has wider VPI
  copy.VCI     = this.VCI;
  copy.CLP     = this.CLP;
  copy.PT      = this.PT;
```

```
  copy.HEC     = this.HEC;
  copy.Payload = this.Payload;
  return copy;
endfunction : to_NNI


// Generate the syndrome array, which is used to compute HEC
function void UNI_cell::generate_syndrome();
   bit [7:0] sndrm;
   for (int i = 0; i < 256; i = i + 1 ) begin
      sndrm = i;
      repeat (8) begin
         if (sndrm[7] === 1'b1)
           sndrm = (sndrm << 1) ^ 8'h07;
         else
           sndrm = sndrm << 1;
      end
      syndrome[i] = sndrm;
   end
   syndrome_not_generated = 0;
endfunction : generate_syndrome


// Compute the HEC value for this object
function bit [7:0] UNI_cell::hec (bit [31:0] hdr);
   hec = 8'h00;
   repeat (4) begin
      hec = syndrome[hec ^ hdr[31:24]];
      hdr = hdr << 8;
   end
   hec = hec ^ 8'h55;
endfunction : hec
```

The `NNI_cell` class is almost identical to `UNI_cell`, except that it does not have a GFC field, or a method to convert to a `UNI_cell`.

Sample 11.18 shows the UNI cells random atomic generator, as originally shown in Section 8.2. The generator randomizes the blueprint instance of the UNI cell, and then sends out a copy of the cell to the driver.

Sample 11.18 UNI_generator class

```
class UNI_generator;
  UNI_cell blueprint; // Blueprint for generator
  mailbox  gen2drv;   // Mailbox to driver for cells
  event    drv2gen;   // Event from driver when done with cell
  int      nCells;    // Num cells for this generator to create
  int      PortID;    // Which Rx port are we generating?

  function new(input mailbox gen2drv,
               input event drv2gen,
               input int nCells, PortID);
    this.gen2drv = gen2drv;
    this.drv2gen = drv2gen;
    this.nCells  = nCells;
    this.PortID  = PortID;
    blueprint = new();
  endfunction : new

  task run();
    UNI_cell cell;
    repeat (nCells) begin
      assert(blueprint.randomize());
      $cast(cell, blueprint.copy());
      cell.display($psprintf("@%0t: Gen%0d: ", $time, PortID));
      gen2drv.put(cell);
      @drv2gen;// Wait for driver to finish with it
    end
  endtask : run

endclass : UNI_generator
```

Sample 11.19 shows the Driver class that sends UNI cells into the ATM switch. This class uses the driver callbacks in Sample 11.20. Note that there is a circular relationship here. The `Driver` class has a queue of `Driver_cbs` objects, and the `pre_tx()` and `post_tx()` methods in `Driver_cbs` are passed `Driver` objects. When you compile the two classes, you may need either `typedef class Driver`, before the `Driver_cbs` class definition, or `typedef class Driver_cbs`, before the `Driver` class definition.

Sample 11.19 driver class

```
typedef class Driver_cbs;

class Driver;

  mailbox gen2drv;     // For cells sent from generator
  event   drv2gen;     // Tell generator when I am done with cell
  vUtopiaRx Rx;        // Virtual ifc for transmitting cells
```

```
  Driver_cbs cbsq[$];  // Queue of callback objects
  int PortID;

  extern function new(input mailbox gen2drv,
                      input event drv2gen,
                      input vUtopiaRx Rx,
                      input int PortID);
  extern task run();
  extern task send (input UNI_cell cell);

endclass : Driver


// new(): Construct a driver object
function Driver::new(input mailbox gen2drv,
                     input event drv2gen,
                     input vUtopiaRx Rx,
                     input int PortID);
  this.gen2drv = gen2drv;
  this.drv2gen = drv2gen;
  this.Rx      = Rx;
  this.PortID  = PortID;
endfunction : new


// run(): Run the driver.
// Get transaction from generator, send into DUT
task Driver::run();
  UNI_cell cell;
  bit drop = 0;

  // Initialize ports
  Rx.cbr.data  <= 0;
  Rx.cbr.soc   <= 0;
  Rx.cbr.clav  <= 0;

  forever begin
    // Read the cell at the front of the mailbox
    gen2drv.peek(cell);
    begin: Tx
      // Pre-transmit callbacks
      foreach (cbsq[i]) begin
        cbsq[i].pre_tx(this, cell, drop);
        if (drop) disable Tx; // Don't transmit this cell
      end

    cell.display($psprintf("@%0t: Drv%0d: ", $time, PortID));
    send(cell);
```

```
    // Post-transmit callbacks
    foreach (cbsq[i])
      cbsq[i].post_tx(this, cell);
    end : Tx

    gen2drv.get(cell);  // Remove cell from the mailbox
    ->drv2gen;  // Tell the generator we are done with this cell
   end
endtask : run


// send(): Send a cell into the DUT
task Driver::send(input UNI_cell cell);
  ATMCellType Pkt;

  cell.pack(Pkt);
  $write("Sending cell: ");
  foreach (Pkt.Mem[i])
  $write("%x ", Pkt.Mem[i]); $display;

  // Iterate thru bytes of cell
  @(Rx.cbr);
  Rx.cbr.clav <= 1;
  for (int i=0; i<=52; i++) begin
    // If not enabled, loop
    while (Rx.cbr.en === 1'b1) @(Rx.cbr);

    // Assert Start Of Cell, assert enable, send byte 0 (i==0)
    Rx.cbr.soc  <= (i == 0);
    Rx.cbr.data <= Pkt.Mem[i];
    @(Rx.cbr);
  end
  Rx.cbr.soc <= 'z;
  Rx.cbr.data <= 8'bx;
  Rx.cbr.clav <= 0;
endtask
```

Sample 11.20 shows the driver callback class, which has simple callbacks that are called before and after a cell is transmitted. This class has empty tasks, which are used by default. A testcase can extend this class to inject new behavior in the driver without having to change any code in the driver

Sample 11.20 Driver callback class

```
typedef class Driver;

class Driver_cbs;
  virtual task pre_tx(input Driver drv,
                      input UNI_cell cell,
                      inout bit drop);
  endtask : pre_tx

  virtual task post_tx(input Driver drv,
                       input UNI_cell cell);
  endtask : post_tx
endclass : Driver_cbs
```

The `Monitor` class has a very simple callback, with just one task that is called after a cell is received.

Sample 11.21 Monitor callback class

```
typedef class Monitor;

class Monitor_cbs;
   virtual task post_rx(input Monitor drv,
                        input NNI_cell cell);
   endtask : post_rx
endclass : Monitor_cbs
```

Sample 11.22 shows the `Monitor` class. Like the `Driver` class, this uses a `typedef` to break the circular compile dependency with `Monitor_cbs`.

Sample 11.22 The Monitor class

```
typedef class Monitor_cbs;

class Monitor;

  vUtopiaTx Tx;          // Virtual interface with output of DUT
  Monitor_cbs cbsq[$];   // Queue of callback objects
  int PortID;

  extern function new(input vUtopiaTx Tx, input int PortID);
  extern task run();
  extern task receive (output NNI_cell cell);
endclass : Monitor


// new(): construct an object
function Monitor::new(input vUtopiaTx Tx, input int PortID);
  this.Tx     = Tx;
```

```
  this.PortID = PortID;
endfunction : new


// run(): Run the monitor
task Monitor::run();
  NNI_cell cell;

  forever begin
    receive(cell);
    foreach (cbsq[i])
      cbsq[i].post_rx(this, cell);  // Post-receive callback
  end
endtask : run


// receive(): Read cell from the DUT, pack into a NNI cell
task Monitor::receive(output NNI_cell cell);
  ATMCellType Pkt;

  Tx.cbt.clav <= 1;
  while (Tx.cbt.soc !== 1'b1 && Tx.cbt.en !== 1'b0)
    @(Tx.cbt);
  for (int i=0; i<=52; i++) begin
    // If not enabled, loop
    while (Tx.cbt.en !== 1'b0) @(Tx.cbt);

    Pkt.Mem[i] = Tx.cbt.data;
    @(Tx.cbt);
  end

  Tx.cbt.clav <= 0;

  cell = new();
  cell.unpack(Pkt);
  cell.display($psprintf("@%0t: Mon%0d: ", $time, PortID));
endtask : receive
```

The scoreboard gets expected cells from the driver through the function `save_expected`, and the cells actually received by the monitor with the function `check_actual`. The function `save_expected()` is called from the callback `Scb_Driver_cbs::post_tx()`, shown in Sample 11.7. The function `check_actual`() is called from `Scb_Monitor_cbs::post_rx()` in Sample 11.8.

Sample 11.23 The Scoreboard class

```
class Expect_cells;
  NNI_cell q[$];
  int iexpect, iactual;
endclass : Expect_cells


class Scoreboard;
  Config cfg;
  Expect_cells expect_cells[];
  NNI_cell cellq[$];
  int iexpect, iactual;

  extern function new(Config cfg);
  extern virtual function void wrap_up();
  extern function void save_expected(UNI_cell ucell);
  extern function void check_actual(input NNI_cell cell,
                                    input int portn);
  extern function void display(string prefix="");
endclass : Scoreboard


function Scoreboard::new(Config cfg);
  this.cfg = cfg;
  expect_cells = new[NumTx];
  foreach (expect_cells[i])
    expect_cells[i] = new();
endfunction : Scoreboard


function void Scoreboard::save_expected(UNI_cell ucell);
  NNI_cell ncell = ucell.to_NNI;
  CellCfgType CellCfg = top.squat.lut.read(ncell.VPI);

  $display("@%0t: Scb save: VPI=%0x, Forward=%b",
           $time, ncell.VPI, CellCfg.FWD);
  ncell.display($psprintf("@%0t: Scb save: ", $time));

  // Find all Tx ports where this cell will be forwarded
  for (int i=0; i<NumTx; i++)
    if (CellCfg.FWD[i]) begin
      expect_cells[i].q.push_back(ncell); // Save cell in this q
      expect_cells[i].iexpect++;
      iexpect++;
    end
endfunction : save_expected
```

```
function void Scoreboard::check_actual(input NNI_cell cell,
                                       input int portn);
  NNI_cell match;
  int match_idx;

  cell.display($psprintf("@%0t: Scb check: ", $time));

  if (expect_cells[portn].q.size() == 0) begin
    $display("@%0t: ERROR: %m cell not found, SCB TX%0d empty",
             $time, portn);
    cell.display("Not Found: ");
    cfg.nErrors++;
    return;
  end

  expect_cells[portn].iactual++;
  iactual++;

  foreach (expect_cells[portn].q[i]) begin
    if (expect_cells[portn].q[i].compare(cell)) begin
      $display("@%0t: Match found for cell", $time);
      expect_cells[portn].q.delete(i);
      return;
    end
  end

  $display("@%0t: ERROR: %m cell not found", $time);
  cell.display("Not Found: ");
  cfg.nErrors++;
endfunction : check_actual


// Print end of simulation report
function void Scoreboard::wrap_up();
  $display("@%0t: %m %0d expected cells, %0d actual cells rcvd",
           $time, iexpect, iactual);

  // Look for leftover cells
  foreach (expect_cells[i]) begin
    if (expect_cells[i].q.size()) begin
 $display("@%0t: %m cells remain in SCB Tx[%0d] at end of test",
          $time, i);
    this.display("Unclaimed: ");
    cfg.nErrors++;
  end
   end
endfunction : wrap_up
```

```
// Print the contents of the scoreboard, mainly for debugging
function void Scoreboard::display(string prefix);
  $display("@%0t: %m so far %0d expected cells, %0d actual
rcvd", $time, iexpect, iactual);
  foreach (expect_cells[i]) begin
    $display("Tx[%0d]: exp=%0d, act=%0d",
          i, expect_cells[i].iexpect, expect_cells[i].iactual);
  foreach (expect_cells[i].q[j])
    expect_cells[i].q[j].display(
                $psprintf("%sScoreboard: Tx%0d: ", prefix, i));
  end
endfunction : display
```

Sample 11.24 shows the class used to gather functional coverage. Since the coverage only looks at data in a single class, the cover group is defined and instantiated inside the `Coverage` class. The data values are read by the class's `sample()` method, then the cover group's `sample()` method is called to record the values.

Sample 11.24 Functional coverage class

```
class Coverage;
  bit [1:0] src;
  bit [NumTx-1:0] fwd;

  covergroup CG_Forward;
    coverpoint src
      {bins src[] = {[0:3]};
       option.weight = 0;}
    coverpoint fwd
      {bins fwd[] = {[1:15]}; // Ignore fwd==0
       option.weight = 0;}
    cross src, fwd;
  endgroup : CG_Forward

  function new;
    CG_Forward = new;    // Instantiate the covergroup
  endfunction : new

  // Sample input data
  function void sample(input bit [1:0] src,
                       input bit [NumTx-1:0] fwd);
    $display("@%0t: Coverage: src=%d. FWD=%b", $time, src, fwd);
    this.src = src;
    this.fwd = fwd;
    CG_Forward.sample();
  endfunction : sample
endclass : Coverage
```

Sample 11.25 shows the `CPU_driver` class that contains the methods to drive the CPU interface.

Sample 11.25 The CPU_driver class

```
class CPU_driver;
  vCPU_T mif;
  CellCfgType lookup [255:0]; // copy of look-up table
  Config cfg;
  bit [NumTx-1:0] fwd;

  extern function new(vCPU_T mif, Config cfg);
  extern task Initialize_Host ();
  extern task HostWrite (int a, CellCfgType d); // configure
  extern task HostRead (int a, output CellCfgType d);
  extern task run();
endclass : CPU_driver


function CPU_driver::new(vCPU_T mif, Config cfg);
  this.mif = mif;
  this.cfg = cfg;
endfunction : new


task CPU_driver::Initialize_Host ();
  mif.BusMode <= 1;
  mif.Addr <= 0;
  mif.DataIn <= 0;
  mif.Sel <= 1;
  mif.Rd_DS <= 1;
  mif.Wr_RW <= 1;
endtask : Initialize_Host


task CPU_driver::HostWrite (int a, CellCfgType d); // configure
  #10 mif.Addr <= a; mif.DataIn <= d; mif.Sel <= 0;
  #10 mif.Wr_RW <= 0;
  while (mif.Rdy_Dtack!==0) #10;
  #10 mif.Wr_RW <= 1; mif.Sel <= 1;
  while (mif.Rdy_Dtack==0) #10;
endtask : HostWrite


task CPU_driver::HostRead (int a, output CellCfgType d);
  #10 mif.Addr <= a; mif.Sel <= 0;
  #10 mif.Rd_DS <= 0;
  while (mif.Rdy_Dtack!==0) #10;
  #10 d = mif.DataOut; mif.Rd_DS <= 1; mif.Sel <= 1;
```

```
  while (mif.Rdy_Dtack==0) #10;
endtask : HostRead


task CPU_driver::run();
  CellCfgType CellFwd;
  Initialize_Host();

  // Configure through Host interface
  repeat (10) @(negedge clk);
  $write("Memory: Loading ... ");
  for (int i=0; i<=255; i++) begin
    CellFwd.FWD = $urandom();
`ifdef FWDALL
    CellFwd.FWD = '1
`endif
    CellFwd.VPI = i;
    HostWrite(i, CellFwd);
    lookup[i] = CellFwd;
  end

  // Verify memory
  $write("Verifying ...");
  for (int i=0; i<=255; i++) begin
    HostRead(i, CellFwd);
    if (lookup[i] != CellFwd) begin
    $display("FATAL, Mem Loc 0x%x contains 0x%x, expected 0x%x",
                  i, lookup[i], CellFwd);
      $finish;
    end
  end
  $display("Verified");

endtask : run
```

11.3 Alternate Tests

The simplest test program is shown in Sample 11.2 and runs with very few constraints. During verification, you will be creating many tests, depending on the major functionality to be tested. Each test can then be run with different seeds.

11.3.1 Your First Test - Just One Cell

The first test you run should probably have just one cell, such as the test in Sample 11.26. You can add a new constraint to the `Config` class by extending it, and then

injecting a new object into the environment before randomization. Once this test works, you can try two cells, and then unconstrain the number of cells to run longer sequences.

Sample 11.26 Test with one cell

```
program automatic test
  #(parameter int NumRx = 4, parameter int NumTx = 4)
   (Utopia.TB_Rx Rx[0:NumRx-1],
    Utopia.TB_Tx Tx[0:NumTx-1],
    cpu_ifc.Test mif,
    input logic rst, clk);

`include "environment.sv"
  Environment env;

class Config_1_cell extends Config;
  constraint one_cells {nCells == 1; }

  function new(input int NumRx,NumTx);
    super.new(NumRx,NumTx);
  endfunction : new
endclass : Config_1_cells


  initial begin
    env = new(Rx, Tx, NumRx, NumTx, mif);

    begin // Just simulate for 1 cell
      Config_1_cells c1 = new(NumRx,NumTx);
      env.cfg = c1;
    end

    env.gen_cfg();    // Config will have just 1 cell
    env.build();
    env.run();
    env.wrap_up();
  end

endprogram // test
```

11.3.2 Randomly Drop Cells

The next test you may run creates errors by occasionally dropping cells, as shown in Sample 11.27. You need to make a callback for the driver that sets the drop bit. Then, in the test, inject this new functionality after the driver class has been constructed during the build phase.

Sample 11.27 Test that drops cells using driver callback

```
program automatic test
  #(parameter int NumRx = 4, parameter int NumTx = 4)
   (Utopia.TB_Rx Rx[0:NumRx-1],
    Utopia.TB_Tx Tx[0:NumTx-1],
    cpu_ifc.Test mif,
    input logic rst, clk);

`include "environment.sv"
  Environment env;

class Driver_cbs_drop extends Driver_cbs;
  virtual task pre_tx(input ATM_cell cell, ref bit drop);
    // Randomly drop 1 out of every 100 transactions
    drop = ($urandom_range(0,99) == 0);
  endtask
 endclass


  initial begin
    env = new(Rx, Tx, NumRx, NumTx, mif);
    env.gen_cfg();
    env.build();

    begin             // Create error injection callback
      Driver_cbs_drop dcd = new();
      env.drv.cbs.push_back(dcd); // Put into driver's Q
    end

    env.run();
    env.wrap_up();
  end

endprogram // test
```

11.4 Conclusion

This chapter shows how you can build a layered testbench, following the guidelines in this book. You can then create new tests by just modifying a single file and injecting new behavior, utilizing the hooks such as callbacks and multiple environment phases.

The testbench was able to get to 100% functional coverage of the ATM switch, at least for the basic cover group. You can use this example to explore more about SystemVerilog testbenches.

Chapter 12

Interfacing with C

In Verilog, you can communicate with C routines using the Programming Language Interface. With the three generations of the PLI (TF, ACC, and VPI routines), you can create delay calculators, connect and synchronize multiple simulators, and add debug tools such as waveform displays. However, the PLI's greatest strength is also its greatest weakness. If you just want to connect a simple C routine using the PLI, you need to write dozens of lines of code, and understand many different concepts such as synchronizing with multiple simulation phases, call frames, and instance pointers. Additionally, the PLI adds overhead to your simulation as it copies data between the Verilog and C domains, in order to protect Verilog data structures from corruption.

SystemVerilog introduces the Direct Programming Interface (DPI), an easier way to interface with C, C++, or any other foreign language. Once you declare or "import" the C routine with the `import` statement, you can call it as if it were any SystemVerilog routine. Additionally, your C code can call SystemVerilog routines. With the DPI you can connect C code that reads stimulus, contains a reference model, or just extends SystemVerilog with new functionality.

If you have a SystemC model that does not consume time, and that you want to connect to SystemVerilog, you can use the DPI. SystemC models with time-consuming methods are best connected with the utilities built into your favorite simulator.

The first half of this chapter is data-centric and shows how you can pass different data types between SystemVerilog and C. The second half is control centric, showing how you can pass control back and forth between SystemVerilog and C.

12.1 Passing Simple Values

The first few examples in this chapter show you how to pass integral values between SystemVerilog and C, and the mechanics of how to declare routines and their arguments on both sides. Later sections show how to pass arrays and structures.

12.1.1 Passing Integer and Real Values

The most basic data type that you can pass between SystemVerilog and C is an `int`, the 2-state, 32-bit type. Sample 12.1 shows the SystemVerilog code that calls a C factorial routine, shown in Sample 12.2.

Sample 12.1 SystemVerilog code calling C factorial routine

```
import "DPI-C" function int factorial(input int i);

program automatic test;
  initial begin
    for (int i=1; i<=10; i++)
      $display("%0d! = %0d", i, factorial(i));
  end
endprogram
```

The `import` statement declares that SystemVerilog routine `factorial` is implemented in a foreign language such as C or C++. The modifier "`DPI-C`" specifies that this is a DPI routine, and the rest of the statement describes the routine arguments.

Sample 12.1 passes 32-bit signed values using the SystemVerilog `int` data type that maps directly to the C `int` type[1]. The C function in Sample 12.2 takes an integer as an input and so the DPI passes the argument by value.

Sample 12.2 C factorial function

```
int factorial(int i) {
  if (i<=1) return 1;
  else      return i*factorial(i-1);
}
```

12.1.2 The Import Declaration

The `import` declaration defines the prototype of the C task or function, but using SystemVerilog types. A C function with a return value is mapped to a SystemVerilog function. A void C function can be mapped to a SystemVerilog task or void function.

[1]The SystemVerilog int is always 32 bits, which the width of an int in C is operating system dependant.

If the name of the C function conflicts with a SystemVerilog name, you can import it with a new name in the SystemVerilog space. In Sample 12.3, the C function `test` is given the SystemVerilog name `my_test`. The C function `expect` is mapped to the SystemVerilog name `fexpect`, as the name `expect` is a reserved keyword in SystemVerilog. The name `expect` becomes a global symbol, used to link with the C code, whereas `fexpect` is a local SystemVerilog symbol. You cannot overload a routine, for example by importing `expect` once with a `real` argument and once with an `int`.

Sample 12.3 Changing the name of an imported function

```
program automatic test;

  // Change name of C function "test" to "my_test"
  import "DPI-C" test = function void my_test();
  initial my_test();

  // C function has same name as keyword, change it
  import "DPI-C" \expect = function int fexpect();
  ...
    if (actual != fexpect()) $display("ERROR");
  ...
endprogram
```

You can import routines anywhere in your SystemVerilog code where you can declare a routine including inside programs, modules, interfaces, packages, and `$unit`, the compilation-unit space. The imported routine will be local to the declaration space in which it is declared. If you need to call an imported method in several locations in your code, put the `import` statement in a package which you import where it is needed. Any changes to the `import` statements are localized to the package.

12.1.3 Argument Directions

Imported C routines can have zero or more arguments. By default the argument direction is `input` (data goes from SystemVerilog to C), but can also be `output` and `inout`. The direction `ref` is not supported. A function can return a simple value such as an integer or real number, or have no return value if you make it `void`.

Sample 12.4 Argument directions

```
import "DPI-C" function int addmul (input int a, b,
                                    s         output int sum,
import "DPI-C" function void stop_model();
```

You can reduce bugs in your C code by declaring any input arguments as `const` and so the C compiler will give an error for any write to an input.

Sample 12.5 C factorial routine with const argument

```
int factorial(const int i) {
  if (i<=1) return 1;
  else      return i*factorial(i-1);
}
```

12.1.4 Argument Types

Each variable that is passed through the DPI has two matching definitions: one for the SystemVerilog side, and one for the C side. It is your responsibility to use compatible types. The SystemVerilog simulator cannot compare the types as it is unable to read the C code. (The VCS simulator produces `vc_hdrs.h` with the C header for any routine that you have imported. You can use this file as a guide to matching the types.)

Table 12-1 shows the data type mapping between SystemVerilog and the inputs and outputs of C routines. The C structures are defined in the include file `svdpi.h`. Arrays mapping is discussed in Sections 12.4 and 12.5, and structures are discussed in Section 12.6.

Table 12-1 Data types mapping between SystemVerilog and C

SystemVerilog	C (input)	C (output)
byte	char	char*
shortint	short int	short int*
int	int	int*
longint	long long int	long int*
shortreal	float	float*
real	double	double*
string	const char*	char**
string[N]	const char**	char**
bit	svBit or unsigned char	svBit* or unsigned char
logic, reg	svLogic or unsigned char	svLogic* or unsigned char*
bit[N:0]	const svBitVecVal*	svBitVecVal*
reg[N:0] logic[N:0]	const svLogicVecVal*	svLogicVecVal*
open array[]	const svOpenArrayHandle	svOpenArrayHandle
chandle	const void*	void*



Note that some mappings are not exact. For example, a `bit` in SystemVerilog maps to `svBit` in C, which ultimately maps to `unsigned char` in the `svdpi.h` include file. As a result, you could write illegal values into the upper bits.

The LRM limits imported function results "small values," which include `void`, `byte`, `shortint`, `int`, `longint`, `real`, `shortreal`, `chandle`, and `string`, plus single bit values of type `bit` and `logic`. A function cannot return a vector such as `bit [6:0]` as this would require returning a pointer to a `svBitVecVal` structure.

12.1.5 Importing a Math Library Routine

Sample 12.6 shows how you can call many functions in the C math library directly, without a C wrapper, thereby reducing the amount of code that you need to write. The Verilog `real` type maps to a C `double`.

Sample 12.6 Importing a C math function

```
import "DPI-C" function real sin(input real r);
...
initial $display("sin(0)=%f", sin(0.0));
```

12.2 Connecting to a Simple C Routine

Your C code might contain a simulation model, such as a processor, that is instantiated side by side with Verilog models. Or your code could be a reference model that is compared to a Verilog model at the transaction or cycle level. Much of this chapter shows an 7-bit counter written in C or C++. Though very simple, the counter has the same parts as a complex model, with inputs, outputs, storage of internal values between calls, and the need to support multiple instances.

12.2.1 A Counter with Static Storage

Sample 12.7 shows the C code for a 7-bit counter, using a static variable to hold the count value.

Sample 12.7 Counter method using a static variable

```
#include <svdpi.h>

void counter7(svBitVecVal *o,
              const svBitVecVal *i,
              const svBit reset,
              const svBit load) {
  static unsigned char count = 0; // Static count storage

  if (reset)     count = 0;    // Reset
  else if (load) count = *i;   // Load value
  else           count++;      // Count
  count &= 0x7f;               // Mask upper bit

  *o = count;
}
```

The `reset` and `load` signals are 2-state single bit signals, and so they are passed as `svBit`, which reduces to `unsigned char`. The input `i` is 2-state, and 7 bits wide, and is passed as `svBitVecVal`. Notice that it is passed as a `const` pointer, which means the underlying value can change, but you cannot change the value of the pointer, such as making it point to another value. Likewise, the `reset` and `load` inputs are also marked as `const`. In this example, the 7-bit counter value is stored in a char, and so you have to mask off the upper bit.

The file `svdpi.h` contains the definitions for SystemVerilog DPI structures and methods. The C code examples in the rest of this chapter leave off the `#include` statements, unless they are important to the discussion.

Sample 12.8 shows a SystemVerilog program that imports and calls the C function for the 7-bit counter.

Sample 12.8 Testbench for an 7-bit counter with static storage

```
import "DPI-C" function void counter7(output bit [6:0] out,
                                      input bit [6:0] in,
                                      input bit reset, load);

program automatic counter;
  bit [6:0] out, in;
  bit       reset, load;

  initial begin
    $monitor("SV: out=%3d, in=%3d, reset=%0d, load=%0d\n",
             out, in,reset, load);
    reset = 0;
    load = 0;
    in = 126;
    out = 42;
    counter7(out, in, reset, load); // Apply default values

    #10 reset = 1;
    counter7(out, in, reset, load); // Apply reset
    ...
  end
endprogram
```

12.2.2 The Chandle Data Type

The `chandle` data type allows you to store a C or C++ pointer in your SystemVerilog code. A `chandle` variable is wide enough to hold a pointer on the machine where the code was compiled, i.e. 32- or 64-bits.

The counter in Sample 12.7 works well as long as it is the only one in the design. The counter value is stored in a static, and so when you instantiate a second counter, it will overwrite the value. If you need more than one instance of a C mode, the C code needs to store its variables somewhere other than a static variable. A better way is to allocate storage, and pass a handle to it, along with the input and output signals values. Sample 12.9 shows a counter that stores the 7-bit count in the structure `c7`. This is overkill for a simple counter, but if you are creating a model for a larger device, you can build from this example.

Sample 12.9 Counter method using instance storage

```
#include <svdpi.h>
#include <malloc.h>
#include <veriuser.h>

typedef struct {  // Structure to hold counter value
  unsigned char cnt;
} c7;

// Construct a counter structure
void* counter7_new() {
  c7* c = (c7*) malloc(sizeof(c7));
  c->cnt = 0;
  return c;
}

// Run the counter for one cycle
void counter7(c7 *inst,
              svBitVecVal* count,
              const svBitVecVal* i,
              const svBit reset,
              const svBit load) {

  if (reset)     inst->cnt = 0;  // Reset
  else if (load) inst->cnt = *i; // Load value
  else           inst->cnt++;    // Count
  inst->cnt &= 0x7f;             // Mask upper bit

  *count = inst->cnt;            // Write to output
  io_printf("C: count=%d, i=%d, reset=%d, load=%d\n",
            *count, *i, reset, load);
}
```

There is a new method, `counter7_new`, to construct the counter instance. This returns a `chandle` that must be passed into the call to `counter7`.

The C code uses the PLI task `io_printf` to display debug messages. The routine is useful when you are debugging C and SystemVerilog code side-by-side as it writes to the same outputs as `$display`, including the simulator's log file. The routine is defined in `veriuser.h`.

The testbench for this counter differs from the static one in several ways. First, the counter must be constructed before it can be used. Next, the counter is called on a clock edge, rather than calling it in-line with the stimulus. For simplicity, the counter is invoked when the clock goes high, and stimulus is applied when the clock goes low, to avoid any race conditions.

Sample 12.10 Testbench for an 7-bit counter with per-instance storage

```
import "DPI-C" function chandle counter7_new();
import "DPI-C" function void counter7
       (input chandle inst,
        output bit [6:0] out,
        input  bit [6:0] in,
        input  bit  reset, load);

program automatic test;

  // Test two instances of the counter
  initial begin
    bit [6:0] o1, o2, i1, i2;
    bit       reset, load, clk1;
    chandle   inst1, inst2;     // Points to storage in C

    inst1 = counter7_new();
    inst2 = counter7_new();
    fork
      forever #10 clk1 = ~clk1;
      forever @(posedge clk1) begin
        counter7(inst1, o1, i1, reset, load);
        counter7(inst2, o2, i2, reset, load);
      end
    join_none

    reset = 0;
    load = 0;
    i1 = 120;
    i2 = 10;

    @(negedge clk1);
    load = 1;

    @(negedge clk1);
    load = 0;
    ...
  end
endprogram
```

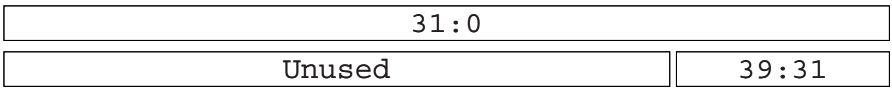
12.2.3 Representation of Packed Values

The string "`DPI-C`"[2] specifies that you are using the canonical representation of packed values. This representation stores a SystemVerilog variable as a C array of one

[2]Earlier versions of the LRM used "DPI" but this is now obsolete and should not be used.

or more elements. A 2-state variable is stored using the type `svBitVecVal`. A 2-state array is stored with multiple elements of this type.

Figure 12-1 Storage of a 40-bit 2-state variable



For performance reasons. the SystemVerilog simulator may not mask the upper bits after calling a DPI routine, and so the SystemVerilog variable could be corrupted. Make sure your C code treats these values properly.

If you need to convert between bits and words, use the macro `SV_PACKED_DATA_NELEMS`. For example, to convert 40 bits to two 32-bit words (as seen in Figure 12-1), use `SV_PACKED_DATA_NELEMS(40)`.

12.2.4 4-State Values

Each 4-state bit in SystemVerilog is stored in the simulator using two bits known as `aval` and `bval`[3], as shown in Table 12-2

Table 12-2 4-state bit encoding

4-State value	aval	bval
0	0	0
1	1	0
Z	0	1
X	1	1

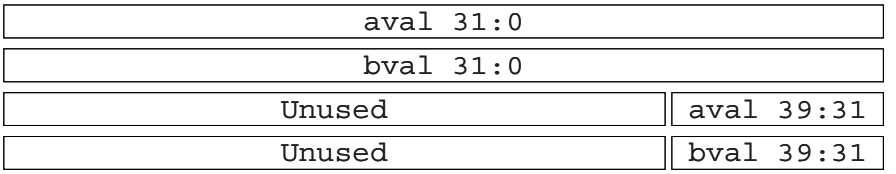
A single bit 4-state variable, such as `logic f`, is stored in an unsigned byte, with the `aval` bit stored in the least significant bit, and the `bval` in the next higher bit. So the value 1'b0 is seen as 0×0 in C, 1'b1 is 0×1, 1'bz is 0×2, and 1'bx is 0×3.

A 4-state vector such as `logic [31:0] lword` is stored using pairs of 32 bits, `svLogicVecVal`, which contains the `aval` and `bval` bits. The 32-bit variable `lword` is stored in a single `svLogicVecVal`. Variables wider than 32-bits are stored in multiple `svLogicVecVal` elements, with the first element containing the 32 least significant bits, and the next element containing the next 32 bits, up to the most significant bits. A 40-bit `logic` variable is stored as one `svLogicVecVal` for the least significant 32 bits, and a second for the upper 8 bits (Figure 12-2). The unused 24-bits in this upper value are undetermined, and you are responsible for masking or extend-

[3]Note that the P1800-2005 LRM has the wrong definitions for vpi_vecval. Instead of a, and b, they should be aval and bval, just like the PLI/VPI. The 2008 version of the LRM has the proper definitions.

ing the sign bit, as needed. The `svLogicVecVal` type is equivalent to `s_vpi_vecval`, which is used to represent 4-state types such as `logic` in the VPI (Verilog Programming Interface).

Figure 12-2 Storage of a 40-bit 4-state variable





Beware of arguments declared without bit subscripts or those declared with a single bit. An argument declared as `input logic a` is stored in an `unsigned char`. The argument `input logic [0:0] b` is `svLogicVecVal`, even though it contains only a single bit.

Sample 12.11 shows the import statements for a 4-state counter.

Sample 12.11 Testbench for counter that checks for Z or X values

```
import "DPI-C" function chandle counter7_new();
import "DPI-C" function void counter7
       (input chandle inst,
        output logic [6:0] out,
        input  logic [6:0] in,
        input  logic  reset, load);
```

The counter previously shown in Sample 12.9 assumes all the inputs are 2-state. Sample 12.12 extends this code to check for Z and X values on `reset`, `load`, and `i`. The actual count is still kept as a 2-state value.

Sample 12.12 Counter method that checks for Z and X values

```
// 4-state replacement for counter7 from Sample 12.9
void counter7(c7 *inst,
              svLogicVecVal* count,
              const svLogicVecVal* i,
              const svLogic reset,
              const svLogic load) {

  if (reset & 0x2) {  // Check just the bval bit of scalar
    io_printf("Error: Z or X detected on reset\n\n");
    return;
  }
  if (load & 0x2) {  // Check just the bval bit of scalar
    io_printf("Error: Z or X detected on load\n\n");
    return;
  }
  if (i->bval) {     // Check just the bval bits of 7-bit vector
    io_printf("Error: Z or X detected on i\n\n");
    return;
  }

  if (reset)     inst->cnt = 0;       // Reset
  else if (load) inst->cnt = i->aval; // Load value
  else           inst->cnt++;         // Count
  inst->cnt &= 0x7f;                  // Mask upper bit

  count->aval = inst->cnt;            // Write to output
  count->bval = 0;
}
```

If you want to force the simulation to terminate cleanly because of a condition found in an imported method, you can call the VPI method `vpi_control(vpiFinish, 0)`. This method and constant are defined in the include file `vpi_user.h`. The value `vpiFinish` tells the simulator to execute a `$finish` after your imported method returns.

12.2.5 Converting from 2-State to 4-State

If you have a DPI application that works with 2-state types and you want to convert it to work with 4-state types, follow the following guidelines.

On the SystemVerilog side, change the import declaration from using 2-state types such as `bit` and `int` to 4-state types such as `logic` and `integer`. Make sure you are using 4-state variables in the function call.

On the C side, switch the argument declarations from `svBitVecVal` to `svLogicVecVal`. Any reference to the arguments will have to use the `.aval` suffix to

correctly access the data. When you read from a 4-state variable, check the `bval` bits to see if there are any Z or X values. When you write to a 4-state variable, clear the `bval` bits unless you need to write Z or X values.

12.3 Connecting to C++

You can use the DPI to connect routines written in C or C++ to SystemVerilog. There are several ways your C++ code can communicate using the DPI, depending on your model's level of abstraction.

12.3.1 The Counter in C++

Sample 12.13 shows a C++ class for the 7-bit counter, with 2-state inputs. It connects to the SystemVerilog testbench in Sample 12.10 and the C++ wrapper code in Sample 12.14.

Sample 12.13 Counter class

```
class Counter7 {
public:
  Counter7();
  void counter7_signal(svBitVecVal* count,
                       const svBitVecVal* i,
                       const svBit reset,
                       const svBit load);
private:
  unsigned char cnt;
};

Counter7::Counter7() {
  cnt = 0;              // Initialize counter
}

void Counter7::counter7_signal(svBitVecVal* count,
                               const svBitVecVal* i,
                               const svBit reset,
                               const svBit load) {
  if (reset)     cnt = 0;     // Reset
  else if (load) cnt = *i;    // Load
  else           cnt++;       // Count
  cnt &= 0x7F;                // Mask upper bit
  *count = cnt;
}
```

12.3.2 Static Methods

The DPI can only call a static C or C++ method, that is, one that is known at link time. As a result, your SystemVerilog code cannot call a C++ method in an object as the object does not exist when the linker runs.

The solution, as shown in Sample 12.14, is to create static methods that can communicate with the C++ dynamic objects and methods. The first method, `counter7_new`, constructs an object for the counter and returns a handle to the object. The second static method, `counter7`, calls the C++ method that performs the counter logic, using the object handle.

Sample 12.14 Static methods and linkage

```
extern "C" void* counter7_new()
{
  return new Counter7;
}

// Call a counter instance, passing the signal values
extern "C" void counter7(void* inst,
                         svBitVecVal* count,
                         const svBitVecVal* i,
                         const svBit reset,
                         const svBit load)
{
  Counter7 *c7 = (Counter7 *) inst;
  c7->counter7_signal(count, i, reset, load);
}
```

The `extern "C"` code tells the C++ compiler that the external information sent to the linker should use C calling conventions and not perform name mangling. You can put this before each method that is called by SystemVerilog, or put `extern "C" { ... }` around a set of methods.

From the testbench point of view, the C++ counter looks the same as the counter that stored the value in per-instance storage, shown in Sample 12.9, and so you can use the same testbench, Sample 12.10, for both.

12.3.3 Communicating with a Transaction Level C++ Model

The previous C/C++ code examples were low-level models that communicated with the SystemVerilog at the signal level. This is not efficient; for example, the counter is called every clock cycle, even if the data or control inputs have not changed. When you create models for complex devices such as processors and networking devices, communicate with them at the transaction level for faster simulations.

The C++ counter model in Sample 12.15 has a transaction-level interface, communicating with methods instead of signals and a clock.

Sample 12.15 C++ counter communicating with methods

```
class Counter7 {
public:
  Counter7();
  void count();
  void load(const svBitVecVal* i);
  void reset();
  int get();
private:
  unsigned char cnt;
};

Counter7::Counter7() {          // Initialize counter
  cnt = 0;
}

void Counter7::count() {        // Increment counter
  cnt = cnt + 1;
  cnt &= 0x7F;                  // Mask upper bit
}

void Counter7::load(const svBitVecVal* i) {
  cnt = *i;
  cnt &= 0x7F;                  // Mask upper bit
}

void Counter7::reset() {
  cnt = 0;
}

// Get the counter value in a pointer to a svBitVecVal
int Counter7::get() {
  return cnt;
}
```

The dynamic C++ methods such as `reset`, `load`, and `count` are wrapped in static methods that use the object handle, passed from SystemVerilog, as shown in Sample 12.16.

Sample 12.16 Static wrapper for C++ transaction level counter

```
#ifdef __cplusplus
extern "C" {
#endif

void* counter7_new() {
  return new Counter7;
}

void counter7_count(void* inst){
  Counter7 *c7 = (Counter7 *) inst;
  c7->count();
}

void counter7_load(void* inst, const svBitVecVal* i) {
  Counter7 *c7 = (Counter7 *) inst;
  c7->load(i);
}

void counter7_reset(void* inst) {
  Counter7 *c7 = (Counter7 *) inst;
  c7->reset();
}

int counter7_get(void* inst) {
  Counter7 *c7 = (Counter7 *) inst;
  return c7->get();
}

#ifdef __cplusplus
}
#endif
```

The OOP interface for the transaction level counter is carried up to the testbench. Sample 12.17 has the SystemVerilog import statements and a class to wrap the C++ object. This allows you to hide the C++ handle inside the class. Note that the `get()` function returns an `int` (32-bit, signed) rather than `bit [6:0]`, as the latter would require returning a pointer to a `svBitVecVal`, as shown in Table 12-1. An imported function can not return a pointer, only return a small value such as `void`, `byte`, `shortint`, `int`, `longint`, `real`, `shortreal`, `chandle`, and `string`, plus single bit values of type `bit` and `logic`.

Sample 12.17 Testbench for C++ model using methods

```
import "DPI-C" function chandle counter7_new();
import "DPI-C" function void counter7_count(input chandle inst);
import "DPI-C" function void counter7_load(input chandle inst,
                                           input bit [6:0] i);
import "DPI-C" function void counter7_reset(input chandle inst);
import "DPI-C" function int counter7_get(input chandle inst);

// Wrap the counter interface with a class
// to hide the C++ instance handle
class Counter7;
   chandle inst;

   function new;
      inst = counter7_new();
   endfunction

   function void count();
      counter7_count(inst);
   endfunction

   function void load(bit [6:0] val);
      counter7_load(inst, val);
   endfunction

   function void reset();
      counter7_reset(inst);
   endfunction

   function bit [6:0] get();
      return counter7_get(inst);
   endfunction

endclass : Counter7
```

Sample 12.18 Testbench for C++ model using methods

```
program automatic counter;
  Counter7 c1;

  initial begin
    c1 = new;

    c1.reset();
    $display("SV: Post reset: counter1=%0d", c1.get());

    c1.load(126);
    if (c1.get() != 126) $display("Error in load");

    c1.count();  // count = 127
    c1.count();  // count = 0
    if (c1.get() != 0) $display("Error in rollover");
   end

endprogram
```

12.4 Simple Array Sharing

So far you have seen examples of passing scalar and vectors between SystemVerilog and C. A typical C model might read an array of values, perform some computation, and return another array with the results.

12.4.1 Single Dimension Arrays – 2-State

Sample 12.19 shows a routine that computes the first 20 values in the Fibonacci series. It is called by the SystemVerilog code in Sample 12.20.

Sample 12.19 C routine to compute Fibonacci series

```
void fib(svBitVecVal data[20]) {
  int i;
  data[0] = 1;
  data[1] = 1;
  for (i=2; i<20; i++)
    data[i] = data[i-1] + data[i-2];
}
```

Sample 12.20 Testbench for Fibonacci routine

```
import "DPI-C" function void fib
                        (output bit [31:0] data[20]);

program automatic test;
  bit [31:0] data[20];

  initial begin
    fib(data);
    foreach (data[i]) $display(i,,data[i]);
   end
endprogram
```

Notice that the array of Fibonacci values is allocated and stored in SystemVerilog, even though it is calculated in C. There is no way to allocate an array in C and reference it in SystemVerilog.

12.4.2 Single Dimension Arrays – 4-State

Sample 12.21 shows the Fibonacci C routine for a 4-state array.

Sample 12.21 C routine to compute Fibonacci series with 4-state array

```
void fib(svLogicVecVal data[20]) {
  int i;
  data[0].aval = 1;    // Write to both aval
  data[0].bval = 0;    //     and bval
  data[1].aval = 1;
  data[1].bval = 0;
  for (i=2; i<20; i++) {
    data[i].aval = data[i-1].aval + data[i-2].aval;
    data[i].bval = 0;  // DonÕt forget to clear bval
  }
}
```

Sample 12.22 Testbench for Fibonacci routine with 4-state array

```
import "DPI-C" function void fib(output logic [31:0] data[20]);

program automatic test;
  logic [31:0] data[20];

  initial begin
    fib(data);
    foreach (data[i]) $display(i,,data[i]);
   end
endprogram
```

Section 12.2.5 describes how to convert a 2-state application to 4-state.

12.5 Open arrays

When sharing arrays between SystemVerilog and C, you have two options. For the fastest simulations, you can reverse-engineer the layout of the elements in SystemVerilog, and write your C code to use this mapping. This approach is fragile, meaning that you will have to rewrite and debug your C code if any of the array sizes change. A more robust approach is to use "open arrays," and their associated SystemVerilog routines to manipulate them. These allow you to write generic C routines that can operate on any size array.

12.5.1 Basic Open Array

Samples 12.23 and 12.24 show how to pass a simple array between SystemVerilog and C with open arrays. Use the empty square brackets `[]` in the SystemVerilog `import` statement to specify that you are passing an open array.

Sample 12.23 Testbench code calling a C routine with an open array

```
import "DPI-C" function void fib_oa(output bit [31:0] data[]);

program automatic test;
  bit [31:0] data[20], r;

  initial begin
    fib_oa(data);
    foreach (data[i])
      $display(i,,data[i]);
   end
endprogram
```

Your C code references the open array with a handle of type `svOpenArrayHandle`. This points to a structure with information about the array such as the declared word range. You can locate the actual array elements with calls such as `svGetArrayPtr`.

Sample 12.24 C code using a basic open array

```
void  fib_oa(const svOpenArrayHandle data_oa) {
  int i, *data;
  data = (int *) svGetArrayPtr(data_oa);
  data[0] = 1;
  data[1] = 1;
  for (i=2; i<=20; i++)
    data[i] = data[i-1] + data[i-2];
}
```

12.5.2 Open Array Methods

There are many DPI methods to access their contents and ranges, as defined in `svdpi.h`. These only work with open array handles declared as `svOpenArrayHandle`, not with pointers such as `svBitVecVal` or `svLogicVecVal`. The following methods give you information about the size of an open array.

Table 12-3 Open array query functions

Function	Description
`int svLeft(h, d)`	Left bound for dimension d
`int svRight(h, d)`	Right bound for dimension d
`int svLow(h, d)`	Low bound for dimension d
`int svHigh(h, d)`	High bound for dimension d
`int svIncrement(h, d)`	1 if left >= right, and –1 if left < right
`int svSize(h, d)`	Number of elements in dimension d: svHigh–svLow+1
`int svDimension(h)`	Number of dimensions in open array
`int svSizeOfArray(h)`	Total size of array in bytes

In Table 12-3, the variable `h` is a `svOpenArrayHandle` and `d` is an `int`.

The following functions return the locations of the C storage for the entire array or a single element (Table 12-4).

Table 12-4 Open array locator functions

Function	Returns pointer to:
`void *svGetArrayPtr(h)`	Storage for the entire array
`void *svGetArrElemPtr(h, i1, ...)`	An element in the array
`void *svGetArrElemPtr1(h, i1)`	An element in a 1-D array
`void *svGetArrElemPtr2(h, i1, i2)`	An element in a 2-D array
`void *svGetArrElemPtr3(h, i1, i2, i3)`	An element in a 3-D array

12.5.3 Passing Unsized Open Arrays

In Sample 12.24, the C code assumed that the array had five elements, numbered 0.4. Sample 12.25 calls C code with a 2-dimensional array. The C code uses the `svLow` and `svHigh` methods to find the array ranges, which, in this example, don't follow the usual 0..size-1.

Sample 12.25 Testbench calling C code with multidimensional open array

```
import "DPI-C" function void mydisplay(inout int h[][]);

program automatic test;
   int a[6:1][8:3];      // Note word ranges are high:low
   initial begin
      foreach (a[i,j]) a[i][j] = i+j;
      mydisplay(a);
      foreach (a[i,j]) $display("V: a[%0d][%0d] = %0d",
                                i, j, a[i][j]);
   end
endprogram
```

This calls the C code in Sample 12.26 that reads the array using the open array methods. The method `svLow(handle, dimension)` returns the lowest index number for the specified dimension. So `svLow(h, 1)` returns 1 for the array declared with the range [6:1]. Likewise, `svHigh(h, 1)` returns 6. You should use `svLow` and `svHigh` with C `for`-loops.

The methods `svLeft` and `svRight` return the left and right index from the array declaration, 6 and 1 respectively for the range [6:1]. At the center of Sample 12.26, the call `svGetArrElemPtr2` returns a pointer to an element in a two dimensional array.

Sample 12.26 C code with multidimensional open array

```
void mydisplay(const svOpenArrayHandle h) {
  int i, j;
  int lo1 = svLow(h, 1);
  int hi1 = svHigh(h, 1);
  int lo2 = svLow(h, 2);
  int hi2 = svHigh(h, 2);
  for (i=lo1; i<=hi1; i++) {
    for (j=lo2; j<=hi2; j++) {
      int *a = (int*) svGetArrElemPtr2(h, i, j);
      io_printf("C: a[%d][%d] = %d\n", i, j, *a);
      *a = i * j;
    }
  }
}
```

12.5.4 Packed Open Arrays in DPI

An open array in the DPI is treated as having a single packed dimension and one or more unpacked dimensions. You can pass an array with multiple packed dimensions, as long as they pack into an element that is the same size as a single element of the formal argument. For example, if you have the formal argument `bit[63:0] b64[]` in the `import` statement, you could pass in the actual argument `bit [1:0][0:3][6:-1] bpack [9:1]`.

Sample 12.27 Testbench for packed open arrays

```
import "DPI-C" function void view_pack(input bit [63:0] b64[]);

program automatic test;
  bit [1:0][0:3][6:-1] bpack[9:1];

  initial begin
    foreach(bpack[i]) bpack[i] = i;
    bpack[2] = 64'h12345678_90abcdef;

    $display("SV: bpack[2]=%h", bpack[2]);  // 64 bits
    $display("SV: bpack[2][0]=%h", bpack[2][0]); // 32 bits
    $display("SV: bpack[2][0][0]=%h", bpack[2][0][0]); // 8 bits

    view_pack(bpack);
  end
endprogram : test
```

Sample 12.28 C code using packed open arrays

```
void view_pack(const svOpenArrayHandle h) {
  int i;

  for (i=svLow(h,1); i<svHigh(h,1); i++)
    io_printf("C: b64[%d]=%llx\n",
              i, *(long long int *)svGetArrElemPtr1(h, i));
}
```

Notice that the C code in Sample 12.28 prints a 64-bit value using `%llx`, and casts the result from `svGetArrayElemPtr1` to `long long int`.

12.6 Sharing Composite Types

By this point you may wonder how to pass objects between SystemVerilog and C. The layout of class properties does not match between the two languages, and so you cannot share objects directly. Instead, you must create similar structures on each side, plus pack and unpack methods to convert between the two formats. Once you have all this in place, you can share composite types.

12.6.1 Passing Structures Between SystemVerilog and C

The following example shares a simple structure for a pixel made of three bytes packed into a word. Sample 12.29 shows the C structure. Notice that C treats a `char` as signed variable, which can give you unexpected results, and so the structure marks the char as unsigned. The bytes are in reverse order from the SystemVerilog because this code was written for a Intel ×86 processor that is little-endian, which means that the least significant byte is stored at a lower address than the most significant. A Sun SPARC is big endian, and so the bytes are stored in the same order as in SystemVerilog: `r`, `b`, `g`.

Sample 12.29 C code to share a structure

```
typedef struct {
  unsigned char b, g, r;  // x86 big-endian
//unsigned char r, g, b;  // SPARC format
} *p_rgb;

void invert(p_rgb rgb) {
  rgb->r = ~rgb->r;     // Invert the color values
  rgb->g = ~rgb->g;
  rgb->b = ~rgb->b;
  io_printf("C:  Invert rgb=%02x,%02x,%02x\n",
            rgb->r, rgb->g, rgb->b);
}
```

The SystemVerilog testbench has a packed struct that holds a single pixel, and class to encapsulate the pixel operations. The `RGB_T` struct is packed and so SystemVerilog will store the bytes in consecutive locations. Without the `packed` modifier, each 8-bit value would be stored in a separate word.

Sample 12.30 Testbench for sharing structure

```
typedef struct packed { bit [ 7:0] r, g, b; } RGB_T;
import "DPI-C" function void invert(inout RGB_T pstruct);

program automatic test;

class RGB;
  rand bit [ 7:0] r, g, b;
  function void display(string prefix="");
    $display("%sRGB=%x,%x,%x", prefix, r, g, b);
  endfunction : display

  // Pack the class properties into a struct
  function RGB_T pack();
    pack.r = r; pack.g = g; pack.b = b;
  endfunction : pack

  // Unpack a struct into the class properties
  function void unpack(RGB_T pstruct);
    r = pstruct.r; g = pstruct.g; b = pstruct.b;
  endfunction : unpack
endclass : RGB

  initial begin
    RGB pixel;
    RGB_T pstruct;

    pixel = new;
    repeat (5) begin
      assert(pixel.randomize());      // Create random pixel
      pixel.display("\nSV: before "); // Print it
      pstruct = pixel.pack();         // Convert to a struct
      invert(pstruct);                // Call C to invert bits
      pixel.unpack(pstruct);          // Unpack struct to class
      pixel.display("SV: after  ");   // Print it
    end
  end
endprogram
```

12.6.2 Passing Strings Between SystemVerilog and C

Using the DPI, you can pass strings from C back to SystemVerilog. You might need to pass a string for the symbolic value of a structure, or get a string representing the internal state of your C code for debug.

The easiest way to pass a string from C to SystemVerilog is for your C function to return a pointer to a static string. The string must be declared as `static` in C, and not as a local string. Nonstatic variables are stored on the stack and are reclaimed when the function returns.

Sample 12.31 Returning a string from C

```
char *print(p_rgb rgb) {
  static char s[12];
  sprintf(s, "%02x,%02x,%02x", rgb->r, rgb->g, rgb->b);
  return s;
}
```

A danger with static storage is that multiple concurrent calls could end up sharing storage. For example, a SystemVerilog `$display` statement that is printing several pixels might call the above `print` method multiple times. Depending on how the SystemVerilog compiler orders these calls, later calls to `print()` could overwrite results from earlier calls, unless the SystemVerilog compiler makes a copy of the string. Note that a call to an imported method can never be interrupted by the SystemVerilog scheduler. Sample 12.32 stores the strings in a heap to support concurrent calls.

Sample 12.32 Returning a string from a heap in C

```
#define PRINT_SIZE 12
#define MAX_CALLS 16
#define HEAP_SIZE PRINT_SIZE * MAX_CALLS

char *print(p_rgb rgb) {
  static char print_heap[HEAP_SIZE + PRINT_SIZE];
  char *s;
  static int heap_idx = 0;
  int nchars;

  s = &print_heap[heap_idx];
  nchars = sprintf(s, "%02x,%02x,%02x",
                   rgb->r, rgb->g, rgb->b);
  heap_idx += nchars + 1;          // DonÕt forget null!
  if (heap_idx > HEAP_SIZE)
    heap_idx = 0;
  return s;
}
```

12.7 Pure and Context Imported Methods

Imported methods are classified as `pure`, `context`, or generic. A `pure` function calculates its output strictly based on its inputs, with no outside interactions. Specifically, a `pure` function does not access any global or static variables, perform any file operations, or interact with anything outside the function such as the operating system, processes, shared memory, sockets, etc. The SystemVerilog compiler may optimize away calls to a `pure` function if the result is not needed, or replace the call with the results from a previous call with the same arguments. The `factorial` function in Sample 12.5, and the `sin` function in 12.6 are both `pure` functions as their result is only based on their inputs.

Sample 12.33 Importing a pure function

```
import "DPI-C" pure function int factorial(input int i);
import "DPI-C" pure function real sin(input real in);
```

An imported method may need to know the context of where it is called so that it can call a PLI TF, ACC, or VPI methods, or a SystemVerilog task that has been exported. Use the `context` attribute for these methods.

Sample 12.34 Imported context tasks

```
import "DPI-C" context task call_sv(bit 31:0] data);
```

An imported method may use global storage, and so it is not `pure`, but might not have any PLI references, and so it does not need the overhead of a `context` method. Sutherland (2004) uses the term "generic" for these methods as the SystemVerilog LRM does not have a specific name. By default, an imported method is generic, as are many of the examples in this chapter.

There is overhead invoking a `context` imported method as the simulator needs to record the calling context, and so only declare a method as `context` if needed. On the other hand, if a generic imported method calls an exported task or a PLI method that accesses SystemVerilog data objects, the simulator could crash.

A context-aware PLI method is one that needs to know where it was called from so that it can access information relative to that location.

12.8 Communicating from C to SystemVerilog

The examples so far have shown you how to call C code from your SystemVerilog models. The DPI also allows you to call SystemVerilog methods from C code. The

SystemVerilog method can be a simple task to record the result from an operation in C, or a time-consuming task representing part of a hardware model.

12.8.1 A Simple Exported Method

Sample 12.35 shows a module that imports a context function, and exports a SystemVerilog function.

Sample 12.35 Exporting a SystemVerilog function

```
module block;
  import "DPI-C" context function void c_display();
  export "DPI-C" function sv_display;  // No type or args

  initial c_display();

  function void sv_display();
    $display("SV: block");
  endfunction
endmodule : block
```



The `export` declaration in Sample 12.35 looks naked because the LRM forbids putting a return value declaration or any arguments. You can't even give the usual empty parentheses. This information in the `export` declaration would duplicate the information in the function declaration at the end of the module and could thus become out of sync if you ever changed the function.

Sample 12.36 shows the C code that calls the exported function.

Sample 12.36 Calling an exported SystemVerilog function from C

```
extern void sv_display();

void c_display() {
  io_printf("C:  c_display\n");
  sv_display();
}
```

This example prints the line from the C code, followed by the `$display` output from the SystemVerilog, as shown in Sample 12.37.

Sample 12.37 Output from simple export

```
C:  c_display
SV: block
```

12.8.2 C Function Calling SystemVerilog Function

While the majority of your testbench should be in SystemVerilog, you may have legacy testbenches in C or other languages, or applications that you want to reuse. This section creates a SystemVerilog memory model that is stimulated by C code that reads transactions from an external file.

The first version of the memory, shown in Samples 12.38 and 12.39, uses only functions, and so everything runs with no elapsed time. The SystemVerilog code calls the C method `read_file` that opens a file. The only command in the file sets the memory size, and so the C code calls an exported function.

Sample 12.38 SystemVerilog module for simple memory model

```
module memory;
  import "DPI-C" function read_file(string fname);
  export "DPI-C" function mem_build;  // No type or args

  initial
    read_file("mem.dat");

  int mem[];

  function mem_build(input int size);
    mem = new[size];     // Allocate dynamic memory elements
  endfunction

endmodule : memory
```

Notice that in Sample 12.38, the `export` statement does not have any arguments as this information is already in the function declaration.

The C code in Sample 12.39 opens the file, reads a command, and calls the exported function. Error checking has been removed for compactness.

Sample 12.39 C code to read simple command file and call exported function

```
extern void mem_build(int);

int read_file(char *fname){
  int cmd;
  FILE *file;

  file = fopen(fname, "r");
  while (!feof(file)) {
    cmd = fgetc(file);
    switch (cmd)
      {
      case 'M': {
        int hi;
        fscanf(file, "%d %d ", &hi);
        mem_build(hi);
        break;
      }
    }
  }
  fclose(file);
}
```

The command file is trivial, with one command to construct a memory with 100 elements.

Sample 12.40 Command file for simple memory model

```
M 100
```

12.8.3 C Task Calling SystemVerilog Task

A real memory model has operations such as read and write that consume time, and thus must be modeled with tasks.

Sample 12.41 shows the SystemVerilog code for the second version of the memory model. It has several improvements compared to Sample 12.38. There are two new tasks, `mem_read` and `mem_write`, which respectively take 20 ns and 10 ns to complete. The imported method `read_file` is now a task as it is calling other tasks. You may remember that only a task can call another task. The `import` statement now specifies that `read_file` is a context method, as the simulator needs to create a separate stack when it is called.

Sample 12.41 SystemVerilog module for memory model with exported tasks

```
module memory;
  import "DPI-C" context task read_file(string fname);
  export "DPI-C" task mem_read;
  export "DPI-C" task mem_write;
  export "DPI-C" function mem_build;

  initial read_file("mem.dat");

  int mem[];

  function mem_build(input int size);
    mem = new[size];
  endfunction

  task mem_read(input int addr, output int data);
    #20 data = mem[addr];
  endtask

  task mem_write(input int addr, input int data);
    #10 mem[addr] = data;
  endtask
endmodule : memory
```

The C code in Sample 12.42 primarily expands the case statement that decodes commands.

Sample 12.42 C code to read command file and call exported function

```
extern void mem_read(int, int*);
extern void mem_write(int, int);
extern void mem_build(int);

int read_file(char *fname) {
  int cmd;
  FILE *file;

  file = fopen(fname, "r");
  while (!feof(file)) {
    cmd = fgetc(file);
    switch (cmd) {
      case 'M': {
        int hi;
        fscanf(file, "%d %d ", &hi);
        mem_build(hi);
        break;
      }
```

```
      case 'R': {
        int addr, data, exp;
        fscanf(file, "%c %d %d ", &cmd, &addr, &data);
        mem_read(addr, &exp);
        if (data != exp)
          io_printf("C: Data=%d, exp=%d\n", data, exp);
        break;
      }

      case 'W': {
        int addr, data;
        fscanf(file, "%c %d %d ", &cmd, &addr, &data);
        mem_write(addr, data);
        break;
      }
    }
  }
  fclose(file);
}
```

The command file has new commands that write two locations, and then reads back one of them, and includes the expected value.

Sample 12.43 Command file for simple memory model

```
M 100
W 12 34
W 99 8
R 12 34
```

12.8.4 Calling Methods in Objects

You can export SystemVerilog methods, except for those defined inside a class. This restriction is similar to the restriction of importing static C methods, as shown in Section 12.3.2, as objects do not exist when SystemVerilog elaborates your code. The solution is to pass a reference to the object between the SystemVerilog and C code. However, unlike a C pointer, a SystemVerilog handle cannot be passed through the DPI. You can instead have an array of handles, and pass the array index between the two languages.

The following examples build on the previous versions of the memory. The SystemVerilog code in Sample 12.45 has a class that encapsulates the memory. Now you can have multiple memories, each in a separate object. The command file in Sample 12.44 creates two memories, M0, and M1. Then it performs several writes to initialized locations in both memories, and lastly tries to read back the values. Notice that location 12 is used for both memories.

Sample 12.44 Command file for exported methods with OOP memories

```
M0 1000
M1 2000
W0 12 34
W1 12 88
W0 99 18
R1 22 44
R0 12 34
R1 12 88
```

The SystemVerilog code constructs a new object for every `M` command in the file. The exported function `mem_build` calls the `Memory` constructor. It then stores the handle to the `Memory` object in a SystemVerilog queue, and returns the queue index to the C code. The handles are stored in a queue so that you can dynamically add new memories. The exported tasks `mem_read` and `mem_write` now have an additional argument, the index of the memory handle in the queue.

Sample 12.45 SystemVerilog module with memory model class

```
module memory;
  import "DPI-C" context task read_file(string fname);
  export "DPI-C" task mem_read;
  export "DPI-C" task mem_write;
  export "DPI-C" function mem_build;

  initial read_file("mem.dat");  // Call C code to read file


  class Memory;
    int mem[];

    function new(input int size);
      mem = new[size];
    endfunction

    task mem_read(input int addr, output int data);
      #20 data = mem[addr];
    endtask

    task mem_write(input int addr, input int data);
      #10 mem[addr] = data;
    endtask : mem_write
  endclass : Memory

  Memory memq[$];          // Queue of Memory objects

  // Construct a new memory instance & push on the queue
  function int mem_build(input int size);
```

```
    Memory m;
    m = new(size);
    memq.push_back(m);
    return memq.size()-1;
  endfunction

  task mem_read(input int idx addr, output int data);
    memq[idx].mem_read(addr, data);
  endtask

  task mem_write(input int idx, addr, input int data);
    memq[idx].mem_write(addr, data);
  endtask

endmodule : memory
```

Sample 12.46 C code to call exported tasks with OOP memory

```
extern void mem_read(int, int, int*);
extern void mem_write(int, int, int);
extern int mem_build(int);

int read_file(char *fname) {
  int cmd, idx;
  FILE *file;

  file = fopen(fname, "r");
  while (!feof(file)) {
    cmd = fgetc(file);
    fscanf(file, "%d", &idx);
    switch (cmd)
      {
      case 'M': {
        int hi, qidx;
        fscanf(file, "%d %d ", &hi);
        qidx = mem_build(hi);
        break;
      }

      case 'R': {
        int addr, data, exp;
        fscanf(file, "%c %d %d ", &cmd, &addr, &data);
        mem_read(idx, addr, &expected);
        if (data != expected)
          io_printf("C: Data=%d, exp=%d\n", data, exp);
        break;
      }
```

```
      case 'W': {
        int addr, data;
        fscanf(file, "%c %d %d ", &cmd, &addr, &data);
        mem_write(idx, addr, data);
        break;
      }
    }
  }
  fclose(file);
}
```

12.8.5 The Meaning of Context

The context of an imported method is the location where it was defined, such as `$unit`, module, program, or package scope, just like a normal SystemVerilog method. If you import a method in two different scopes, the corresponding C code executes in the context of where the `import` statement occurred. This is similar to defining a SystemVerilog `run()` task in each of two separate modules. Each task accesses variables in its own module, with no ambiguity.

If you add a second module to Sample 12.35 that imports the same C code and exports its own function, the C method will call different SystemVerilog methods, depending on the context of the import and export statements.

Sample 12.47 Second module for simple export example

```
module top;
  import "DPI-C" context function void c_display();
  export "DPI-C" function sv_display;

  block b1();
  initial c_display();

  function void sv_display();
    $display("SV: top");
  endfunction
endmodule : top


module block;
  import "DPI-C" context function void c_display();
  export "DPI-C" function sv_display;

  initial c_display();

  function void sv_display();
    $display("SV: block");
```

```
  endfunction
endmodule : block
```

The output shows that one C method calls two separate SystemVerilog methods, depending on where the C method was called.

Sample 12.48 Output from simple example with two modules

```
C:  c_display
SV: block
C:  c_display
SV: top
```

12.8.6 Setting the Scope for an Imported Method

Just as your SystemVerilog code can call a method in the local scope, an imported C method can call a method outside its default context. Use the method `svGetScope` to get a handle to the current scope, and then use that handle in a call to `svSetScope` to make the C code think it is inside another context. Sample 12.49 shows the C code for two methods. The first, `save_my_scope()`, saves the scope of where it was called from the SystemVerilog side. The second method, `c_display()`, sets its scope to the saved one, prints a message, then calls your function, `sv_display()`.

Sample 12.49 C code getting and setting context

```
extern void sv_display();
svScope my_scope;

void save_my_scope() {
  my_scope = svGetScope();
}

void c_display() {
  // Print the current scope
  io_printf("\nC: c_display called from scope %s\n",
            svGetNameFromScope(svGetScope()));

  // Set a new scope
  svSetScope(my_scope);
  io_printf("C: calling %s.sv_display\n",
            svGetNameFromScope(svGetScope()));
  sv_display();
}
```

The C code calls `svGetNameFromScope()` that returns a string of the current scope. The scope is printed twice, once with the scope where the C code was first called from, and again with the scope that was previously saved. The routine `svGetScope`

`FromName()` takes a string with a SystemVerilog scope and returns a pointer to a `svScope` handle that can be used with `svSetScope()`.

In the SystemVerilog code in Sample 12.50, the first module, `block`, calls a C method that saves the context. When the module `top` calls `c_display()`, the method sets scope back to `block`, and so it calls the `sv_display()` method in the `block` module, not the `top` module.

Sample 12.50 Modules calling methods that get and set context

```
module block;
  import "DPI-C" context function void c_display();
  import "DPI-C" context function void save_my_scope();
  export "DPI-C" function sv_display;

  function void sv_display();
    $display("SV: %m");
  endfunction : sv_display

  initial begin
    save_my_scope();
    c_display();
  end

endmodule : block


module top;
  import "DPI-C" context function void c_display();
  export "DPI-C" function sv_display;

  function void sv_display();
    $display("SV: %m");
  endfunction : sv_display

  block b1();

  initial #1 c_display();

endmodule : top
```

This produces the output shown in Sample 12.51.

Sample 12.51 Output from svSetScope code

```
C: c_display called from top.b1
C: Calling top.b1.sv_display
SV: top.b1.sv_display

C: c_display called from top
C: Calling top.b1.sv_display
SV: top.b1.sv_display
```

You could use this concept of scope to allow a C model to know where it was instantiated from, and differentiate each instance. For example, a memory model may be instantiated several times, and needs to allocate unique storage for every instance.

12.9 Connecting Other Languages

This chapter has shown the DPI working with C and C++. With a little work, you can connect other languages. The easiest way is to call the Verilog `$system()` task. If you need the return value from the return value from the command, use the Unix `system()` function and the `WEXITSTATUS` macro. The SystemVerilog code in Sample 12.52 calls a C wrapper for `system()`.

Sample 12.52 SystemVerilog code calling C wrapper for Perl

```
import "DPI-C" function int call_perl(string s);

program automatic perl_test;
   int ret_val;
   string script;

   initial begin
      if (!$test$plusargs("script")) begin
        $display("No +script switch found");
        $finish;
      end
      $value$plusargs("script=%s", script);
      $display("SV: Running '%0s'", script);
      ret_val = call_perl(script);
      $display("SV: Perl script returned %0d", ret_val );
   end
endprogram : perl_test
```

Sample 12.53 is the C wrapper that calls system() and translates the return value.

Sample 12.53 C wrapper for Perl script

```
#include "vc_hdrs.h"
#include <stdlib.h>
#include <wait.h>

int call_perl(const char* command) {
  int result = system(command);
  return WEXITSTATUS(result);
}
```

Sample 12.54 is a Perl script that prints a message and returns a value.

Sample 12.54 Perl script called from C and SystemVerilog

```
#!/usr/local/bin/perl
print "Perl: Hello world!\n" ;
exit (3)
```

12.10 Conclusion

The DPI allows you to call C routines as if they are just another SystemVerilog routine, passing SystemVerilog types directly into C. This has less overhead than the PLI, which builds argument lists, and always has to keep track of the calling context, not to mention the complexity of having up to four C routines for every system task.

Additionally, with the DPI, your C code can call SystemVerilog routines, allowing external applications to control simulation. With the PLI you would need trigger variables and more argument lists, and you have to worry about subtle bugs from multiple calls to time-consuming tasks.

The most difficult part of the DPI is mapping SystemVerilog types to C, especially if you have structures and classes that are shared between the two languages. If you can master this problem, you can connect almost any application to SystemVerilog.

References

Bergeron, Janick. *Writing Testbenches Using SystemVerilog*. Norwell, MA: Springer, 2006

Bergeron, Janick; Cerny, Eduard; Hunter, Alan; and Nightingale, Andrew. *Verification Methodology Manual for SystemVerilog*. Norwell, MA: Springer, 2005

Cohen, Ben; Venkataramanan, Srinivasan; and Kumari, Ajeetha. *SystemVerilog Assertions Handbook for Formal and Dynamic Verification*: VhdlCohen Publishing, 2005

Cummings, Cliff; *Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!* San Jose, CA: Synopsys User Group, 2000

Cummings, Cliff; and Salz, Arturo. *SystemVerilog Event Regions, Race Avoidance & Guidelines*, Synopsys User Group, Boston, CA, 2006

Denning, Peter. *The Locality Principle*, Communications of the ACM, 48(7), July 2005, pp. 19–24

Haque, Faisal, Michelson, Jonathan. *The Art of Verification with SystemVerilog Assertions*. Verification Central, 2006

IEEE. *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*. New York: IEEE, 2005 (a.k.a. SystemVerilog Language Reference Manual, or LRM.)

IEEE. *IEEE Standard Verilog Hardware Design, Description Language*. New York: IEEE, 2001

Patt, Yale N.; and Patel, Sanjay J. *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. New York City, NY: McGraw Hill, 2003

Sutherland, Stuart. *Integrating SystemC Models with Verilog and SystemVerilog Using the SystemVerilog Direct Programing Interface*. Europe; Synopsys User Group, 2004

Sutherland, Stuart; Davidmann, Simon; Flake, Peter; and Moorby, Phil. *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Norwell, MA: Springer, 2006

Sutherland, Stuart; and Mills, Don. *Verilog and SystemVerilog Gotchas*. Norwell, MA: Springer, 2007

Synopsys, Inc. *Hybrid RTL Formal Verification Ensures Early Detection of Corner-Case Bugs*, http://synopsys.com/products/magellan/magellan_wp.html, 2003

van der Schoot, Hans; and Bergeron, Janick *Transaction-Level Functional Coverage in SystemVerilog*. San Jose, CA: DVCon, 2006

Vijayaraghavan, Srikanth, and Ramanathan, Meyyappan. *A Practical Guide for SystemVerilog Assertions.* Norwell, MA: Springer, 2005

Wachowski, Andy; and Wachowski, Larry. *The Matrix*. Hollywood, CA: Warner Brothers Studios, 1999

Index

Symbols

! 171
cycle delay 100, 122, 345
$ 37, 171, 314–315
$cast 58, 270, 272, 277
$dist_exponential 187
$dist_normal 187
$dist_poisson 187
$dist_uniform 187
$error 108
$exit 95, 101, 118
$fatal 108, 165–166
$fclose 64
$feof 40, 64
$finish 95, 101, 118, 392
$fopen 40, 64
$fscanf 40, 64–65
$get_coverage 331
$info 108
$isunknown 27, 109
$psprintf 60, 327
$random 187
$realtime 76–77
$root 105, 141–142
$sformat 60
$size 29–30, 35, 42
$strobe 342
$system 418
$test$plusargs 356–357, 418
$time 27, 76–77
$timeformat 27, 76–77
$unit 104–105, 141, 383, 415
$urandom 187
$urandom_range 42, 187
$value$plusargs 356–357, 418
$warning 108
%0t 27
%m 327
%t 76
+= 64, 70, 145
+ntb_random_seed 356
:: 136, 238
<< 53
>> 53–55, 156
? : operator 31, 43
[$] 36
[*] 38
[] 34
^= 68
`define 49

Numerics

2-state types 27, 130, 382, 386, 390, 392
4-state types 27, 130, 390–392

A

abstract class 282–284, 286
Accellera xxx
accumulate operator 64
Active region 93–94, 96, 98, 102
Agent class 158
always block 218
always blocks in programs 101
anonymous enumerated type 55
arguments

default value 69–70
sticky 67
task and function 66
type 67
array
assignment 35
associative 38–40, 42, 47
compare 31
constraint 195
copy 31
dynamic 34–35, 38, 42, 46–47
fixed-size 28, 35, 38, 42, 46–47
handle 150
linked list 40
literal 29, 36
locator methods 42
methods 41
multidimensional 28, 30–31
packed 33–34
queue 36, 38, 46
reduction methods 41
unpacked 29, 34
assertion
concurrent 108
coverage 300
immediate 107–108
procedural 166
associative array 38–40, 42, 47
at_least option 304, 328
ATM router 109–112, 114–116
atomic stimulus generation 204
auto_bin_max 311–312, 318
auto_bin_max option 327
automatic 68, 74–75, 95, 131–132, 139, 225–226
in threads 224
aval 390, 392

B

backdoor load 106
Backus-Naur Form 205
base class 263
BaseTr 282, 292
begin...end 66, 218
optional in tasks and functions 66
Bergeron, Janick xxxv, 4, 351, 421
bidirectional constraints 176–177
bidirectional signal 100
binsof 322–325
bit data type 27, 385, 396
bit streaming
see streaming operator 53
blueprint pattern 265
BNF 205
bounded mailbox 245–246
break 64
bval 390, 392
byte data type 27, 384–385, 396

C

calc_crc method 261–262, 273
callback
coverage 307
creation 286
inject distrubance 284
scoreboard 288
usage 287
cast 52, 58, 61, 65, 270, 277
$cast 53, 272, 280
function return value 65
Cerny, Eduard 421
chandle data type 385, 387–389, 396
char 384
Checker class 158
class 126, 128
class constructor 129
class scope resolution operator 136, 238
clock generator 102
clocking block 89–91, 95, 100–101, 114–115
Cohen, Ben 421
comment
covergroup 327
comment option 305, 327–328
compilation unit 104
composition 259, 274–276, 278
concatenation
string 59
concurrent assertion 108
conditional operator 31, 43
Config class 255
const type 59, 68, 105, 168
constrained 8
constrained-random test 8–9, 161, 187, 295
constraint

array 195
block 165
dist 169–170, 173, 188, 331
in extended class 270
inside 168, 171–174, 176–178, 195
constraint_mode 182–184, 191
constructor 129, 131, 263, 282
containment 144
context 410–411
continue 64
copy
deep 153
method 153–154, 266–267, 279–280, 282
object 151, 279
shallow 153
copy_data method 280–282
covergroup
comment 327
generic 325–326
option 327
sample 303, 307–308
trigger 307
coverpoint 303–304, 306–307, 309–329
CRC 127
cross coverage 319–325
cross module reference 105, 340–341, 346
cross_num_print_missing option 329
CRT 161, 187, 295
Cummings, Cliff 92, 94, 421
cyclic random 165
cyclic redundancy check 127

D

data type
bit 27
byte 27
int 27
integer 25
logic 26
longint 27
real 25
reg 25–26
shortint 27
time 25
wire 26
DDR 89
deallocation 132
decrement 63
default statement 89
default value 69–70
2-state and 4-state 130
delete method 35, 37, 39, 48
derived class 263
Direct Programming Interface 381
disable 219, 228–232
disable fork 229–230
disable label 230–232
display method 139–140, 261–262, 271
dist constraint 169–170, 173, 188, 331
do...while loop 39, 57, 64
domain 310
double data rate clock 89
double data type 384–385
downcasting 270
DPI 381
DPI-C 382, 389
Driver class 158
dynamic array 34–35, 38, 42, 46–47
dynamic threads 223–224

E

enumerated types 55
enumerated values 56
enumeration 55
Environment class 158
event 233–237, 253
event triggered 233–237, 250
exists 39–40, 48
expression width 60
extended class 263
extern
see external routine declaration 139
external constraint 192–193
external routine declaration 139–141, 255, 283

F

file I/O 64
final block 118
find_first method 43
find_index method 43–44
find_last method 43
find_last_index method 43
first 39

first method 39, 47, 57–58
fixed-size array 28, 35, 38, 42, 46–47
float data type 384
for loop 29–30, 39, 57, 63
force design signals 106
foreach constraint 195, 200–201, 362
foreach loop 29–30, 35, 37, 40
fork...join 218–219
fork...join_any 218, 221–222
fork...join_none 218, 220–223, 226–227
four-state types 27
    see 4-state types 27
function 65
    arguments 66
functional coverage 422
    using callbacks 289

G

garbage collection 132
Generator class 158, 222–223, 236, 238, 243, 265, 267, 292
get_coverage 331
get_inst_coverage 331
getc method 59–60
goal option 329

H

handle 128–129
    array 150
Haque, Faisal 421
Hardware Description Language xxx
Hardware Verification Language xxx, 2
HDL xxx
histogram 172–173
hook 266, 285
HVL xxx–xxxi, 2, 14

I

iff 109, 315
ignore_bins 317–318, 322–323
illegal_bins 318
immediate assertion 107–108
implicit port connection 103
import 382
increment 63
inheritance 260, 274
initialization in declaration 74
in-line constraint 192
inout
    argument type 67
    port type 117
input
    argument type 67
    port type 117
insert queue 37
inside 168, 171
inside constraint 168, 171–174, 176–178, 195
instantiate 129
int data type 27, 384–385, 396
integer data type 25, 27, 51
interface 83, 90, 104, 339
    connecting to port 85
    procedural code 347
    virtual 333–334, 336–338, 341–347
interprocess communication xxxi, 2, 217, 232, 253
intersect 322–323, 325
io_printf 388, 403
IPC
    see interprocess communication 217
iterator argument 43

L

last method 57
LC3 microcontoller 118
linked list 40
local 157, 290–291
logic data type 26–27, 90, 385, 396
long long int 404
longint data type 27, 384–385, 396
LRM 98
LRM SystemVerilog 47, 70, 94, 101, 128, 193, 217, 333, 421

M

macro 48
macromodule 104
Magellan 422
mailbox 240, 246, 251–253
    bounded 245–246
    unbounded 245
makes Jack a dull boy xxxiii
malloc 129
max method 42

method 128, 263
    virtual 272–274, 279–281, 283
min method 42
modport 85, 114
module 127
Monitor class 158
multidimensional array 28, 30–31

N

name function 55, 174
new
    constructor 129–131
    copying objects 151
new function 130
new[] operator 35, 131
next method 39, 47, 57
nonblocking assignment 84, 88, 92–93, 98, 421
null 129, 133, 189
num method 39

O

object 128–129
    copy 151, 279
    deallocation 132
    instantiation 129
Object-Oriented Programming
    see OOP 125
Observed region 94
OOP 125–126
    terminology 128, 263
OOP analogy
    badge 132
    car 126
    house 128
open array 400
Open arrays 400
OpenVera xxx
option
    at_least 304, 328
    auto_bin_max 327
    comment 305, 327–328
    cross_num_print_missing 329
    goal 329
    per_instance 327–328, 331
    weight 323–325, 327
options.auto_bin_max 311–312, 318
or method 41
output
    argument type 67
    port type 117

P

pack 155–156, 404–405
package 49, 59, 104, 127
packed array 33–34
parameter 49, 59, 104
parameterized class 290
parent class 263
Patt, Yale 422
per_instance option 327–328, 331
Perl hash array 39
physical interfaces 333
PLI 381, 388
polymorphism 273–274
pop_back 37
port
    connecting to interface 85
post_randomize 185–186, 203, 205
post-decrement 63
post-increment 63
Postponed 94, 96–97
pre_randomize 185–186
pre-decrement 63
pre-increment 63
prev method 47, 57
primitive 104
PRNG 162, 166, 209–211
procedural assertion 166
process 217
product method 41
program 74, 95, 104, 127, 227, 257
program blocks
    single vs. multiple 95
property 128, 263
prototype 128, 263, 282
pseudo-random number generator 162, 166, 209–210
public 157
pure
    imported method 407
    virtual method 282–284, 286
push_back method 43
push_front 37
putc method 60

Q

queue 36, 38, 42, 46
literal 36–37

R

rand 165
rand_mode 188–189
randc 165, 174, 195, 201
randcase 207–208
random real 167
random seed 12–13, 162, 166, 209, 211
random stability 210–212
randomize function 166–167
randomize() with 184, 192
randomize(null) 189
randsequence 205
Reactive region 94, 96, 98
real data type 25, 27, 51, 76, 167, 384–385, 396
ref 383
argument type 70
port type 117
reference counting 133
reg data type 25–26
return 72
routine arguments 66
run method 265, 267–268

S

s_vpi_vecval 391
sample method 303, 307–308
scenario generation 204
scoreboard 18, 45–46, 158, 285, 287–289
using callbacks 285
semaphore 238–240, 253
shortint data type 27, 384–385, 396
shortreal data type 384–385, 396
signature 274
sine function 385, 407
size function 35, 187, 195
solve...before 180–182, 330–331, 362
sparse matrix 38
start method 315
state variables 189–190
static
method 136–138
storage 74
variable 135–137
stop method 315
streaming operator 53–54, 155–156
string concatenation 59
string data type 59–60, 384–385, 396
struct 50–52, 55
sub class 263
substr method 60
sum method 41, 44, 196
super class 263
Sutherland, Stuart 351, 422
SV_PACKED_DATA_NELEMS 390
SVA 109, 300, 421–422
svBit 384–386, 388, 393–394
svBitVecVal 384–386, 390, 392, 401
svDimension 401
svdpi.h 384, 386, 388, 401
svGetArrayElemPtr1 404
svGetArrayPtr 400–402
svGetArrElemPtr 402
svGetArrElemPtr1 402
svGetArrElemPtr2 402–403
svGetArrElemPtr3 402
svGetNameFromScope 416
svGetScope 416
svGetScopeFromName 416
svHigh 401–403
svIncrement 401
svLeft 401–402
svLogic 384, 392
svLogicVecVal 384, 390–392, 401
svLow 401–403
svOpenArrayHandle 400–401, 403–404
svRight 401–402
svScope 416
svSetScope 416
svSize 401
svSizeOfArray 401
synchronous drive 98
system() 418
SystemC 381
SystemVerilog Assertion 109, 300, 309, 421–422

T

task 65
arguments 66
template - see parameterized class
The Matrix 1, 422

this 143–144
thread 217
time data type 25, 27, 76
time literals 76
timeprecision 75–76
timescale 75–77, 105–106
timeunit 75–76
tolower method 59
toupper method 59–60
transactor 158
transition coverage 317
triggered 235
triggered method 233–237, 250
two-state types
    see 2-state types 27
typedef 48
    array 50, 73
    class 146
    enum 56
    struct 50–52, 55
    union 51
    virtual interface 345

U

uint user-defined type 49, 199
unbounded mailbox 245
union 51
unique method 43
unpack 155–156, 404–405
unpacked array 28–29, 33–34
unsigned 27, 49, 187, 194, 200

V

van der Schoot, Hans 422
vc_hdrs.h 384
Verification Methodology Manual for SystemVerilog
    see VMM 1
Verilog Programming Interface 391
Verilog-1995 xxix–xxx, 25, 28, 32, 66–67, 74, 110, 187, 333
Verilog-2001 xxix–xxx, 28, 32, 59–60, 64, 66, 74, 81, 85, 422
veriuser.h 388
virtual
    class - see abstract class
    interface 333–334, 336–338, 341–347
    memory 273
    method 272–274, 279–281, 283
virtual method 264, 273
VMM 1, 4–5, 18, 90, 206, 265, 300, 341, 421
void data type 65, 385, 396
void function 65, 186
VPI 381, 391
vpi_control 392
vpiFinish 392

W

wait 219, 224, 229, 238, 257
wait fork 227
weight option 323–325, 327
which 232
wildcard 317, 325
wire data type 26, 90
wrap_up method 268

X

XMR 340–341, 346
xor method 41